Universitá degli Studi di Cagliari
Dipartimento di Ingegneria Meccanica
Dottorato di Ricerca in Progettazione Meccanica
XXIII Ciclo - Ing-Ind/08

Thesis for the degree of Doctor of Philosophy

Design and Implementation of a Computational Environment for High Performance Scientific Computing

---

Author:
Alessandro Sanna

Supervisor:
Prof. Chiara Palomba

Doctoral Program coordinator:
Prof. Natalino Mandas

Final exam academic year 2010 - 2011

March 2012

Contact information:
Alessandro SANNA
via Claudio Villa 35
09134 Cagliari
ITALY
email: alessandrosanna83@gmail.com

This manuscript was typeset on LaTeX.
All the research work was done with the UNIX-Family operating system.

Dedicated to
my mother Maurizia,
my father Ignazio,
my sister Alina,
and my girlfriend Aina

There are more things in heaven and earth, Horatio, Than are dreamt of in your philosophy.
Hamlet Act 1, scene 5, 159–167, Shakespeare

Everything make soup.
A. Sanna

Abstract

This thesis has a three-fold aim. The first one regards the design and implementation of a multiphysics framework for high-performance scientific and engineering computing. The other two aims are built on top of the first one, and they regard the development and implementation of two software components for the simulation of Fluid Flow and Radiative Heat Transfer respectively. The thesis is organized in three parts, one for each aim.
The first aim is devoted to the development and implementation of a software platform on top of which it is possible to develop software application aimed to any specific kind of simulation. The keylines of this platform are: effectiveness, efficiency, flexibility and usability. Following these keylines simultaneously is a challenging task, it can be accomplished only by developing novel strategies for making the following three aspects sinergically work together: numerical methods, physical modeling, computer science. In the first part of the thesis we shall discuss our efforts in this direction.
The second aim is devoted to the development and implementation of a Fluid Flow solver able to deal with Euler and Laminar Navier-Stokes equations. It is able to perform 3D parallel computation on mixed unstructured meshes. Therefore an application on top of the multiphysics framework has been developed and implemented which is able to address the problem of simulating flow fields using the finite volume method.
The third aim is devoted to another application, that is, the computation of Radiative Heat Transfer. This aim was accomplished by developing some novel metodologies, such as the development and parallel implementation of a particular Monte Carlo approach for computing radiative heat transfer, and a novel particle tracking algorithm for tracking particle-like entities accross mixed 2D or 3D unstructured meshes in parallel computation.

Key words: Multiphysics Framework, Computational Fluid Dynamics, Radiative Heat Transfer, Monte Carlo Method, Finite Volume Method, Particle Tracking, Random Number Generators, Computer Science, Object Oriented Programming, Parallel Algorithms.

Acknowledgements

I would like to start by thanking my supervisor at the University of Cagliari, Prof. Chiara Palomba, even at distance, she has constantly taken care of my thesis with kindness, profesionality and human qualities.

I would like to thank my supervisor at the von Karman Institute for Fluid Dynamics, Prof. Herman Deconinck, for allowing me to conduct this research work at the von Karman Institute and for promoting this work and assisting me in this research.

A special thanks is devoted to Dr. Andrea Lani, which, during my Diploma Course, was my true teacher in CFD, and transmitted me the passion and curiosity for this science.

I would also like to thank Prof. Thierry Magin, for supervising me during my work on Monte Carlo Method for Radiative Heat Transfer.

I would then like to acknowledge the faculty of the Department of Mechanical Engineering of the University of Cagliari. In particular I am grateful to Prof. Franco Nurzia and Prof. Natalino Mandas.

I also offer my thanks to the faculty, the computer center, the library and all the members of the von Karman Institute *family*: I have really appreciated the special good atmosphere that you have made me live at VKI.

I thanks all my friends and colleagues at the von Karman Institute for the nice time we spent together. In particular I want to thanks István Horváth, Jesus Garicano Mena, Davide Masutti, Antony Delmas, Guillaume Grossir, Henny Bottini, Khalil Bensassi, Fabio Pinna, Alessandro Munafo, Antonino Bonanni, Stefano D'Angelo, Simone Duni, Alberto Sonda and Marco Pau.

Obviously I want to thanks all my three friends-brothers in Cagliari: Nicola Matteo and Nicolas, grazie ragazzi! Voglio ringraziare tanto zia Milvia, zio Renato, le mie due Nonne e tutte le persone che mi vogliono bene per tutto il supporto ed affetto che mi hanno dato.

Muchas gracias to my girlfriend Aina, a really special person, for the all the love, patience and encouragement that you unconditionally give to me.

Grazie infinite to my mother Maurizia, the star of my life, to my father Ignazio and to my sister Alina, for the all the love and strength you have always given to me; thanks to be the family that we are.

Contents

List of Figures

Listings

> The sciences do not try to explain, they hardly even try to interpret, they mainly make models. By a model is meant a mathematical construct which, with the addition of certain verbal interpretations, describes observed phenomena. The justification of such a mathematical construct is solely and precisely that it is expected to work.
>
> John von Neumann

Chapter 1

Introduction

1.1 Background

Computational Science. For centuries sciences have followed the fundamental paradigm of first observing phenomena, then theorizing on them. They finally verify the theory through experiments. Similarly engineering has first design, then building and testing the prototypes and finally build the product.
In the last decades a new tool has emerged to help the development of science and engineering: *simulation*. To *simulate* means that the reality under study is reproduced inside a computer. In order to obtain this reproduction, two ingredients are usually necessary:

- The *discretization* of the laws governing the physics to be simulated.
- A suitable representation of the domain of the independent variables; that is: geometry and time.

Nowadays the execution of detailed simulations is becoming more and more important to replace the execution of complicated and expensive experiments. Therefore observations and experiments, in the scientific paradigm, and design and prototyping, in the engineering paradigm are being progressively substituted by simulations. In the engineering field this is called *virtual prototyping*.

Virtual Prototyping. With *virtual prototyping* it is intended the set of procedures made with computational tools used by engineers to develop a product from the design to the final manufacturing process. Some of the most important types of computational tools are *Computer-Assisted Design* (**CAD**), *Computer-Assisted Manufacturing* (**CAM**) and *Computer-Assisted Engineering* (**CAE**). Many different CAEs exist depending on the physical domain one wants to simulate. Quite important CAEs are *Computational Fluid Dynamics* (**CFD**) and *Computational Solid Mechanics* (**CSM**). In fig. (1.1) is sketched the virtual prototyping cycle.

The virtual prototyping cycle is divided in three phases. In the first phase the shape of the product is designed according to its specifications. In the second phase the shape's performance is tested with simulations. If this performance is not satisfactory, step back to phase 1 and

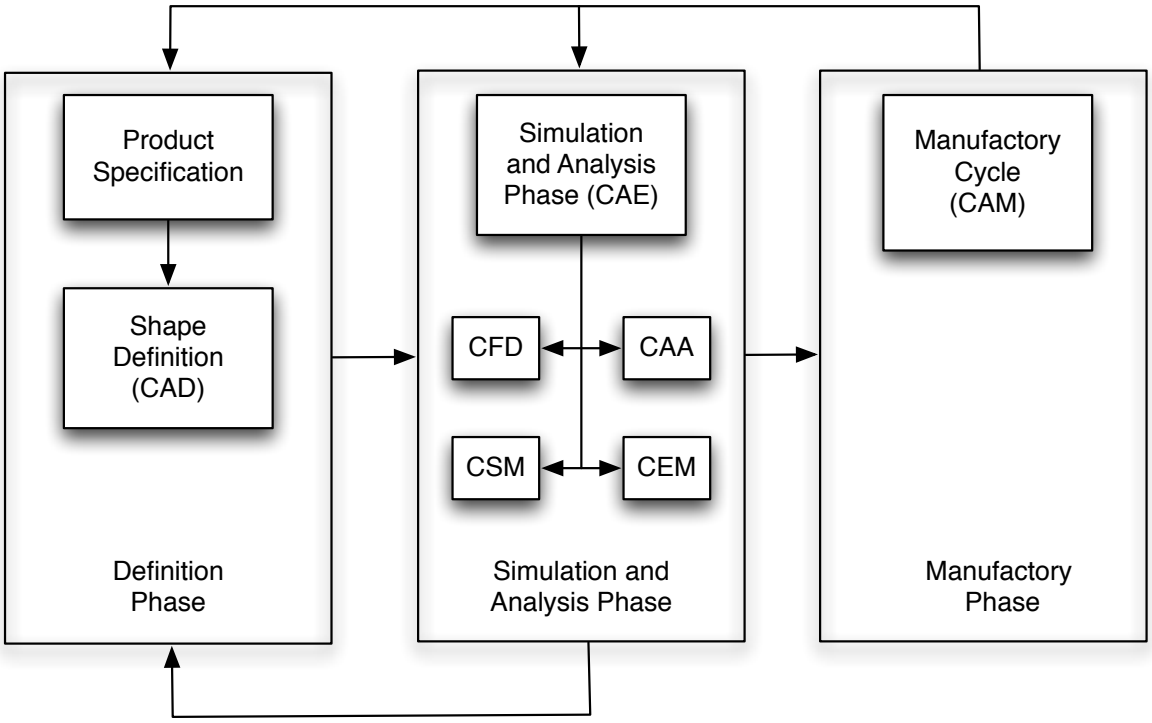


Figure 1.1: Virtual Prototyping Process.

reshape the object. Otherwise the process goes forward to design the manufactory cycle. If the consequent manufactory cycle's cost and the market's response is satisfactory the cycle ends, otherwise the cycle loops back to the first and/or second phase to improve the product's features.

Benefits. There are many obvious benefits in using simulations as substitute of experiments, some of them are:

- Simulating is quite less expensive than setting up an experiment.
- Simulating is less complex than performing an experiment.
- One can easily simulate just the phenomena that is under study and highlight its effects leaving aside sources of noise and unrequired influences.
- Simulating can be quite faster than performing an experiment.
- In some cases it is impossible or extremely difficult to perform an experiment, and simulations are the only tools one has to study a particular phenomenon. As an example, this happens for some astrophysical or combustion phenomena.

Discretization. The process of casting a mathematical model of a physical phenomenon in a form suitable to be implemented in a computer program is called discretization. The mathematical model of a physical phenomenon (one can speak of **Physical Model**) is usually a set of *governing equations* in the form of *Partial Differential Equations* (**PDE**) together with a suitable set of boundary conditions and closure constitutive relations. Many different methods

exist to dicretize the PDEs, among which one can cite: *Finite Difference Method* (**FDM**), *Finite Element Method* (**FEM**), *Finite Volume Method* (**FVM**), *Residual Distribution Scheme* (**RDS**), *Discontinuous Galerkin* (**DG**) and *Spectral Method* (**SM**). Each one of these methods has many different variants (sub-Methods), and each variant is particularly suitable to discretize some particular PDEs, that is: they are tailored for a particular group of physical phenomena. Often the same physical phenomena can be solved by means of different methods, or sub-methods: one then speaks of **Multi-Methods** possibility. Sometimes the problem at hand is made of a set of complex physical phenomena, each one governed by its own Physical Model, in this case one speaks of **Multi-Physics** problems.

Domain Representation. The space domain is often called **Computational Domain**. A computational domain is normally subdivided into control volumes, called elements or cells. The set of all elements that subdivide a computational domain is called **Mesh** or **Grid**. Many different type of Mesh subdivisions exist, mainly one has:

- *Non overlapping structured mesh*: it is suitable only for simple geometries.
- *Chimera mesh*: it is made of a set of partially overlapping structured meshes and it is suitable even for complex geometries.
- *Unstructured mesh*: it is suitable for geometries of arbitrary complexity, and its creation is automatable, but it is more complicated to implement an efficient data structure.

High Performance Computing. Let's introduce some terminology. The thesis distinguishes between an *Application* and a *Computational Environment* (**CE**). We call *Application*, a software system able to simulate a set of one or more (in this case coupled) physical phenomena. Instead we call CE a software tools system for Multi-Physics, Multi-Methods and Unstructured Mesh computations. Appropriately using or combining some of the tools of a CE we can build a variety of applications. Implement an effective and efficient CE is an extremely difficult task. It requires the use of modern programming techniques, such as:

- Object Oriented Programming (OOP) [42].
- Generic Programming (GP) [104].
- Operator Overloading [30].
- Meta Programming Techniques (MP) [112].
- Parallel Programming (PP) [88].

It is quite accepted by the scientific community that in order to properly design and implement a complex CE the old *procedural programming* is not adequate enough. OOP has emerged as the favorite programming paradigm for complex software systems. GP is a programming techniques typical of C++ which allows to write generic algorithm that can operate on generic types of variables, not just built-in data types like double or integer but even quite complex class objects. Operator overloading is another peculiarity of C++ that allow us to overload any operator in order to use it to perform operations between class objects. MP are techniques that basically make a first program to be programmed by a second program, thereby allowing the former to be more optimized than if it had been directly programmed by the programmer. Parallel Programming is used in order to share the simulation task among many processors

which can indeed work together in parallel, allowing the computation to be executed with a speed proportional to their number.

Software Complexity. The above techniques enable the construction of complex software systems, but unfortunatelly they are quite difficult to be learnt and properly used by scientists and engineers for their daily numerical developments. Indeed the usual practice is the use of scientific environment like Matlab [77], Octave [38] and Python [111]. These are really good softwares, but they mainly only allow to build monolithic, not cooperative and not HPC (High Performance Computing) codes, nothing like an effective and efficient CE. An interesting solution would be a CE made of independent HPC modules that:

- are able to cooperate and interact with each other, like independent agents (see Agent Oriented Programming);
- are reusable: their applications are not hard coded;
- interact with each other with a transparent interface: each one use the others as a black box;
- are specialized for a specific task in order to accomplish it the best they can;
- are easy to understand, modify or extend independently;

In this way, building an application for a particular simulation would be as easy as:

- Apply the *Divide and Conquer* paradigm. That is: analyze which tasks are necessary to perform the simulation, and assign a specific software module to each task.
- If we have already implemented a module which we need again, then we just reuse it, otherwise we implement it from scratch.
- Connect all the modules in order to obtain the software for that particular application.

Implementing this paradigm the job of computational scientists and engineers would be easier, faster and better.

Example. Let's consider a CE made of the following modules:

1. Paraview [10] writer;
2. Gambit [5] reader;
3. Unstructured mesh data structure [105];
4. Dependent Variable data structure [105];
5. Mesh partitioner [108];
6. Euler Physical Model [18];
7. Solid Mechanics Physical Model [125];
8. Finite Volume Method [50];
9. Finite Element Method [126].

Than an application for inviscid flow simulation would be made by connecting the modules (1),(2),(3),(4),(5),(6),(8), while an application for solid mechanics simulations would be made by connecting the modules (1),(2),(3),(4),(5),(7),(9). As one can see there are some modules that partecipate only in one or few applications, and there are some others that, if made general enough, can be reused for every application. These are said to be part of the **Kernel** of a CE. Inside a kernel there can usually be found data structure-related modules, like (1),(2),(3),(4),(5) of the above list.

1.2 Objectives

The focus of this thesis are CEs. The objectives are the design and implementation of:

1. A flexible kernel;
2. An application for fluid flow simulations;
3. An application for radiative heat transfer simulations.

In order to tackle the first objective, we have proposed a series of solutions to answer some issues that naturally came out when developing a kernel for scientific computations. For the second objective we have developed and implemented a state of the art unstructured parallel 3D fluid flow solver. Finally for the third objective we have developed a novel parallel unstructured 2D and 3D Monte Carlo Method implementation for radiative heat transfer simulations. Both the second and the third objectives make use of the results obtained with the first objective. Fluid flow and radiative heat transfer are potentially coupleable simulations. In fact if the radiative phenomenon is propagating inside a fluid, and not in the vacuum off-Earth space, then its heat appears as a source term inside the fluid flow equations.
We called the set of these three objective the *HYDRA project*. Hydra in Greec mythology was a "nameless serpent-like chthonic water beast (as its name evinces) that possessed many heads" [6].
There exist some projects that share some aspects with Hydra, but they all have proposed different solutions. Some of them are: OpenFOAM [37], CoolFLUID [59][93], FEniCS [4], LibMesh [55], LiveV [7], OpenFlower [8], Overture [9], etc .. [3].

1.3 Outline

After this introduction the manuscript is divided in four parts. Each one of the firsts three parts is concerned with one of the three objectives of this thesis and is composed of three chapters. Instead the fourth part comprises the conclusion and the bibliography. The manuscript is decomposed into the following chapters.

Part I

Chapter 2 introduces some choices and tools we made for HPC.

Chapter 3 explains the design and implementation of the parallel unstructured 3D mesh and dependent variable data structure.

Chapter 4 explains the design and implementation of an object oriented framework for the solvers.

Part II

Chapter 5 reviews fluid flow modeling methods.

Chapter 6 describes our state of the art finite volume solver.

Chapter 7 summarizes the validation testcases for the finite volume solver.

Part III

Chapter 8 introduces radiative transfer modeling and our approach to tackle it. Our approach consist of using a novel implementation of the Monte Carlo Method that allows to define the statistical pattern of the computational domain before time iterations begin.

Chapter 9 describe our solution for tracking particles inside a computational domain. This solution is needed in order to track the virtual photons emitted by the fluid and solid surfaces of the computational domain.

Chapter 10 summarizes the implementation and validation of our Monte Carlo Method for Radiative Heat Transfer. Here we describe how the code is made and parallelized. Moreover we show the outcome of two testcases for the validation of the whole procedure.

Part IV

Chapter 11 gathers observations, achievments and perspective of the present work.

Bibliography

Part I

Multiphysics Framework

Preamble of Part I

A *kernel* for a computational environment, also called **Multiphysics Framework**, is one of the most difficult pieces of software that can be written. Its successful conception and design requires expertise in three domains:

- physic modeling;
- numerical Mathematics;
- Computer Science.

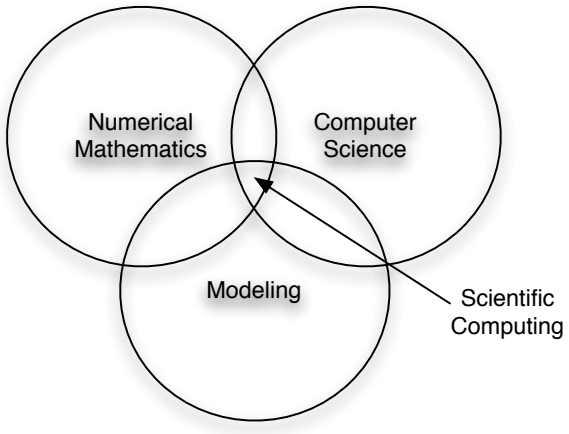


Figure 1.2: Scientific Computing Domains

The interaction of these three domains, each with its own specific features and considerations, gives rise to the final product: a Multiphysics Framework. In contemporary literature there are still no indications of methods and strategies for a proper design of such type of products: most of the notions still belong to the research field. Therefore the conception and development of a multiphysics framework is a really challenging task that one can try to tackle, but without the help of a guide.
In this first part of the thesis, the reserch made on the development of a multiphysics framework is put forward. The thesis' approach consists of three main blocks:

1. Implementation of fundamental choices, and selection and/or development of some basic programming tools.
2. Development of the data structure.

3. Development of the framework for the solvers.

Each of these blocks is dealt with individually in the next three chapters.

Language shapes the way we think, and determines what we can think about.

B.L. Whorf

Programming is understanding.

Kristen Nygaard

Chapter 2

High Performance Computing

2.1 Context

At the very beginning of a software development process some basic choices have to be made; in our context these are:

1. programming language;
2. parallel library;
3. numerical libraries to be pluged into our code;
4. pre-processing software;
5. post-processing software.

The choice of the programming language is fundamental, as it will directly determine things like:

- What we can implement: not everything is implementable with every programming language, and some are more suitable than others for a given task.
- The code's final execution speed.
- The complexity of the implemented entities.
- The complexity of the code's maintenance and extension.
- Parallel issues.
- etc...

As we will explain in paragraph 2.2, we rely on C++ for the programming language.
Concerning the programming library the "de facto" standard in high performance computing (HPC) is the use of the **MPI** library. By definition no HPC exists if it has no parallel capabilities.
In almost every simulation there is a moment where we end up solving a potentially very large linear system. This linear system can stand alone or be part of a sequence of linear systems

for the solution of a non linear problem, as it is the case inside implicit time discretization methods, or optimization algorithms. Some famous open source libraries were developed at the beginning of the 70s for the solution of these systems. Among them one finds **BLAS**, **LAPACK** and **ATLAS**. During the last 30 years they have been improved and tuned for every kind of computer platform so that nowadays people prefer to not reinvent the wheel and plug them inside their own codes. This practice is followed also by most of the commercial CFD software houses.
It is well know that a prerequisite of any numerical simulation is a mesh that represents the computational domains. There are many different good commercial and open source codes that assume the task of building the mesh, in the phase commonly known as *pre-processing phase*. In this thesis we are not going to deal with this phase, instead we will rely on the existing mesh generators, and we shall simply write interfaces to their file formats.
After the execution of the simulation the large amounts of generated data have to be analysed, this is commonly done by visualizing them. For this purpose we rely on commercial and open source programs for which we implement their file format writter programs.
After the aforementioned basic choices, we will describe two basic programming tools we developed that we consider extremely importand and whose use is widespread across the whole multiphysics framework. These tools are: Reference Counted Pointers, and Parameter Lists. In the last paragraphs of this chapter we will address their development.

2.2 Programming Language

We have chosen to use C++ as our programming language because of its unique features that differentiate it from every other language, among which we can mention:

- It allows the production of high-performance native codes;
- It provides support for operator overloading;
- It provides support for object-oriented programming (OOP);
- It provides support for generic programming (GP);
- It is a strong typing language (this contributes to the production of high-performance codes);
- It has a powerful Turing-complete compile-time programming feature: template meta-programming;
- It provides support for the creation of very efficient concrete data types, whose efficiency is comparable to that of built-in data types;
- It provides support for pointers;
- It provides support for dynamic memory allocation;
- There exist extremely performant open source compilers;

To the best of our knowledge there are no other programming languages which offer such powerful set of features. For instance in C++ we can develop new data types that can be used

to develop new programming languages inside C++ itself. That is: we are able to define a new programming language, that could be tuned for a certain application, with a level of efficiency and customizability that could not be achieved with any other programming language.

2.3 OOP and GP in C++

Any programming language can be broken down into two high level concepts:

- Data, and
- Operations on data

Though this may seem like a trivial statement, quite often in science and engineering problems the real work that needs to be done is identifying what the relevant data are, and what operations need to be executed on those data in order to obtain the desired results. Designing a program consists exactly in identifying what data are needed to represent the problem, and what algorithms will be acting on the data. In this paragraph we will concentrate on how this design can be made in the C++ object oriented language (OOP).

2.3.1 Object

Object oriented programs are constituted by objects. An object is a variable that binds both data (also called **members**) and operations (also called **methods**) to data. The set of members of an object represents its internal **state**. An object can execute a method if there is another object that sends it a **request** or **message**. The former object is called **server**, while the later is called **client**. A request is the only possible way to let an object execute a method and thereby change its state.
An object can represent any kind of entity: from a single parameter to a whole framework, from a real physical entity to an abstract mathematical concept. A difficult task in object oriented design is the decomposition of a system in objects. Many factors play a role in this task: encapsulation, granularity, dependency, flexibility, performance, evolution, reusability, etc...
The objects are created instantiating a **class**. A class is the **type** of an object, whereas the object is said to be an **instance** of a class. We like to think on the difference between classes and objects in terms of classical Aristotelic phylosophy: an object is to its class as a substance is to its form. There can be just one class representing the concept of a *Cat*, but there can be many different objects representing concrete cats (all instances of the Cat class). Each instance of a class is individuated by its own state.
An object can instantiate another object of its same or different class. In order to represents this kind of relationships, the *Unified Modeling Language* (**UML**) [13] uses a dashed arrow with "«instantiate»" written over it, as sketched in fig. 2.1:

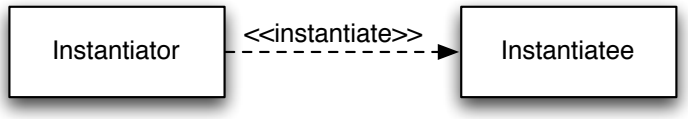


Figure 2.1: Instantiation relationship

2.3.2 Interface

Every method is *declared* inside an object's class. A declaration specifies the name of the method, its parameters and return value. Altogether different from the method's declaration is the method's **definition**. The definition is the concrete implementation of a method. Normally the declarations and the implementations of a class are written in two distinct files, respectivelly: the **header** and the **implementer**. The set of all method declarations referred to a class object is called the **interface**. Therefore the interface defines all requests that can be sent to an object. This means that objects interact with each other through their interfaces; the objects are accessible only through their interfaces. These features determine two fundamental peculiarities of OOP, which are:

- **Data abstraction.** Declaration and definition are separated.
- **Data encapsulation.** The client only knows the declaration of a server's method through the later's interface, yet it knows nothing about the server's definition or state. Therefore the client uses the server as a black box.

Thus we can say that OOP is programming through interfaces, not through implementation.

2.3.3 Inheritance

A class can inherit another class. The former is called the **derived** class, while the later is called the **base** class. The derived class "*is a*" base class, in the same way that a cat *is a* mammal. The derived class has ownership of the interface and eventually also of the members of the base class it has inherited. Moreover the derived class can choose to use the method definition of the base class or it can reimplement it. In this last case we speak of an **overridding** of the base method definition. Two derived classes inheriting from a same base one can have completely different implementations. The derived class can also have members and methods that are not present in the base class. In this case we say that the derived class **specializes** the base class. We define as an **abstract class** a base class that contains only its method declarations, without the definitions. The purpose of the abstract class is to define a common interface for its derived classes.
In UML the inheritance relationship is represented with a vertical line and a triangle, as sketched in fig. 2.2:

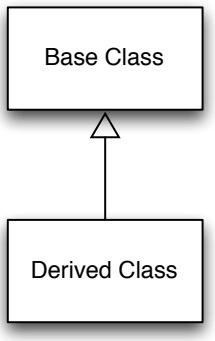


Figure 2.2: Inheritance relationship

2.3.4 Polymorphism

Based on the above, it is clear that:

- A client can send a request to any server that has an interface able to reply (capable servers);
- The capable servers can have different implementations so they can reply in different ways.

If the association between client and server is done at compile time we speak of *static binding*. But if we cannot know in advance which server will reply to the client, then we need to make the association during the execution of the program, in which case we speak of *dynamic binding*. The dynamic binding allows, during execution time, the substitution between objects that have the same interface. This substitutability is called polymorphism.

2.3.5 Composition

An object can be a member of another object. This means that we can assemble or compose objects (**compositors**) in order to obtain another object (**composition**) with more complex functionalities. There are two ways to compose objects:

- **Composition in strict sense.** The composition *owns* its composers and it is responsible for their lifetime.
- **Acquaintance or Aggregation.** The composition only *knows* its composers but it is not responsible for their lifetime.

In fig. 2.3 are sketched the UML representations of these relationships.

The composition can be made at compile time (static composition) or at execution time (dynamic composition). Due to dynamic binding and composition, the compile time structure of an object oriented program turns out to be quite different from the execution time one. The structure of a code consists of classes linked by fixed relationship of inheritance and composition. However at execution time the code's structure results in a complex net of communicating objects that can change dramatically.

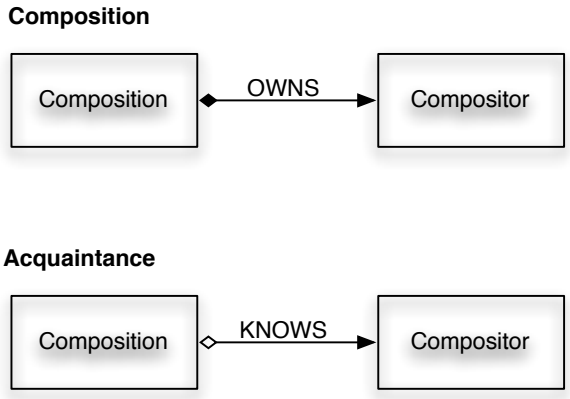


Figure 2.3: Composition relationships

2.3.6 Templates

In order to declare a class we have to specify the classes of its object members and the classes of the objects used as parameters or return value of its methods. This is because C++ requires us to declare variables, functions, and most other kind of entities using specific types. However, many algorithms and data structures look the same regardless of the kind of variables they refer to. For example, a sorting algorithm works in the same way no matter whether it is sorting objects representing integers, cars, or equations. A data structure for storing elements, is identical irrespective of the kind of elements it stores.
In these cases we would like to have classes and functions which operate on variables whose type is a variable itself, and so it can be set as if it were a parameter. Template classes and template functions [112] are exactly the tools that allow us to parametrize the types of other classes and functions. When we use a template we pass the types as arguments, implicitly or explicitly.
In today's programs, templates are widely used. For example, inside the C++ Standard Template Library (STL) almost all codes are template codes. In this thesis we have made extensive use of templates and STL.
In fig. 2.4 is sketched the UML representations of template classes. The template class *List* is a container whose elements' type is parametrized: it can store any type of element. The class *ShapeList* is a List whose type is parametrized with objects of type *shape*. Shape could be a class for representing a geometrical shape.

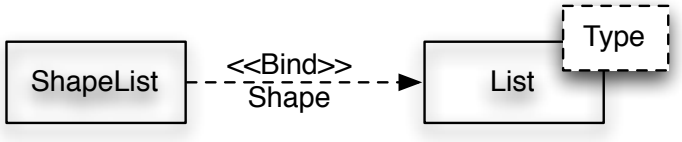


Figure 2.4: Template relationship

2.3.7 Example

In fig. 2.5 is shown an example. Here we have a ShapeList object which is a container of pointers to shape objects. It is built as a template container *List* parametrized with pointers to shape objects. ShapeList has therefore acquaintance of many shape objects. The asterisk by the arrowheads between ShapeList and Shape means that the multiplicity value of Shape objects is above 1: in this concrete example, we have two objects derived from Shape, Polygon and Circle. A Polygon has ownership of at least 3 Points (its vertices), while a Circle has ownership of 1 Point (its center). On the other hand, both Polygon and Circle have acquaintance of 1 Style in which they are drawn. Polygon and Circle are instantiated by an object representing the Drawer.
In the rest of the thesis we shall often make use of this type of diagrams.

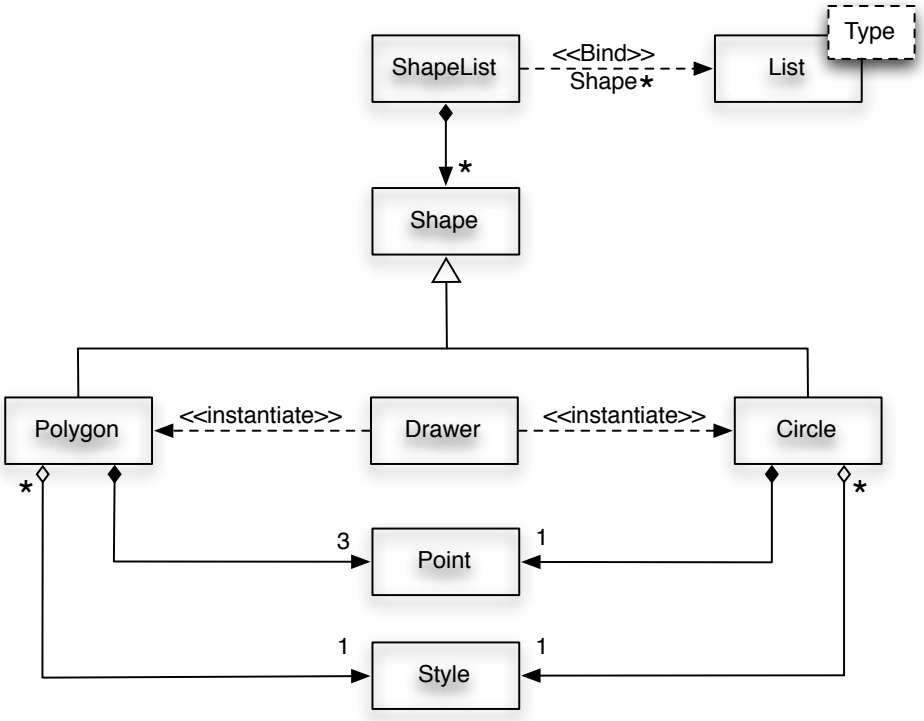


Figure 2.5: Example of object relationships

2.4 MPI

We rely on *Message Passing Interface* (**MPI**) [88],[102] as a mean to write parallel programs. Nowadays MPI is the standard for parallel programming because it provides portability and efficiency, and it has received wide acceptance by both academia and industry.
Let's define as **process** an instance of a program or a sub-program that is being executed, more or less autonomously, on a **physical processor**. MPI is a library made with a set of functions which allows processes running on the same or different physical processors to exchange data.

The basic concept of MPI is that of **message**. A message is the information exchanged between two processes, and it consists of: a content and an envelope. The content specifies the data that we want to transfer between a sender process and a receiver process. The envelope is all information needed to perform this transfer, such as:

- rank of the sending process;
- rank of the receiving process;
- type of data;
- size of data;
- group of processes which the sender and the receiver belong to;
- tag.

Almost everything in MPI is based on the simple idea of "*Message Sent - Message Received*". Indeed all MPI functions can either be:

- means to send or receive messages between two or more processes;
- means to perform set-ups for calling the previous functions.

Learning to use MPI is not an easy task, and writing an effective parallel program is even more difficult. It is important to emphasize that we cannot write a parallel program simply using the data structures and algorithms of serial one. In most cases, the serial algorithms and the serial data structures have to be completely redesigned.

2.5 External Libraries

In scientific computing programs, basic operations with vectors and matrices play a key role. The computational complexity of these operations is measured in reference to the number n of elements of the vector. For example, an inner product between vectors requires n multiplications and $n-1$ additions, that is, a total of approximately $2n$ operations. Therefore the computational complexity of the inner product is $O(n)$ (read as "order n"). Similarly, we can estimate the computational complexity of the outer product to be $O(n^2)$. For a matrix-to-matrix multiplication, the complexity is $O(n^3)$.
Achieving efficient and clean programs with these operations has always been mandatory. In the last few decades there have been several efforts to standardize and optimize such operations. The result of these efforts is **BLAS** [1]. BLAS stands for **B**asic **L**inear **A**lgebra **S**ubprograms, and it was first proposed by Lawson et al. [60] and further developed in subsequent years. BLAS is subdivided in three levels; the first level is for $O(n)$ operations, the second is for $O(n^2)$, while the third is for $O(n^3)$.
BLAS consistently offers the best performance for executing a given type of operation in a given type of computer; this is why computer vendors optimize BLAS for specific architectures. A recent trend has been the development of a new generation of "self-tuning" BLAS to target the memory complexity of modern processors. As examples of this trend we can mention **LAPACK** and **ATLAS**: the **A**utomatically **T**uned **L**inear **A**lgebra **S**oftware. Few further libraries for solving linear systems with specific methods have been recently developed using

BLAS, LAPACK and ATLAS, such as PETSc [57], from the Argonne National Laboratories, and Trilinos [58], from Sandia National Laboratories. We will rely on these two libraries to solve linear systems.

2.6 Reference Counted Pointers

2.6.1 Problem Statement

We present here four situations where we need to use raw C++ pointers, and the problems that can arise out of this use.

Dynamic memory allocations. C++ has dynamic memory allocation capability; this means that we can create objects at execution time. Creating an object means allocating an amount of memory of a size equal to that of the object we want to fit in there. Usually a dynamically allocated object is used only during a relatively short part of a program's execution period. Therefore in order to save memory we almost always need to delete the dynamically allocated object once it is no longer necessary. To allocate an object, we use the *new* operator, while to deallocate it we use the *delete* operator. These two operators have to be used carefully, otherwise they easily lead to crashes or memory leaks. In the particular instance of complex systems of objects, it can be quite difficult to track each dynamically allocated object, and call the operator *delete* in the right moment. Understanding all the reasons to avoid direct calls to operator *delete* is beyond the scope of this thesis (the interested reader can have a look at [16],[81] and [80]), but we must stress the need for a strategy to hide the use of this operator. In fact we need dynamic memory allocation in order to achieve dynamic binding, and the safe passing of large parameters between objects.

Dynamic binding. Dynamic binding lends a code the capability to be configured at execution time. This means that:

- The end-user can configure the code for her/his own purposes. He/She could, for example, choose a particular combination of discretizing methods.
- While it is running, the code has the capability to change the algorithms being used according to the computational needs. For example, if the simulation is not converging, then the code can autonomously decide to change the preconditioning method.

C++ attains dynamic binding allowing pointers to base classes to be assigned to derived class objects. When we carry out this assignment, we need to dynamically allocate the derived object.

Passing parameters. Often we will need to pass a huge object parameter between two other objects. This passage can be accomplished in three ways:

1. Passing the parameter as value.
2. Passing the parameters as a reference.

3. Passing the parameters as a pointer.

In the first method the receiving object has to make a copy of the parameters and save its own copy. This leads to wasted memory for the copied object, and extra time used for the copying operation. The second and third methods avoid the drawbacks of the first one, with the additional consideration that the third one has an advantage with respect to the second one: it can be polymorphic.

Object sharing. In this thesis we shall see that there are many situations where we want some objects, which we call the *owners*, to share another object, which we call the *owned*. All owners aggregate the same owned object, they do not simply have an acquaintance of it. C++ does not have built-in capacity to provide this type of relationship.

2.6.2 Proposed solution

In order to solve all the aforementioned problems, we propose the use of a *referece counted pointer* (**RCP**) in all cases where normally a raw C++ pointer is used. An RCP, [81], is an object that wraps a raw C++ pointer and provides some useful services. In fig. 2.6 is shown the mechanism of our RCP. We can have many RCP objects that have acquaintance of the same Counter object. Basically a Counter is an object which store a pointer to the Object we want to share, as well as the number of RCPs pointing to it and consequently to the shared object. Both RCP and Counter are templatized with the class of the shared object. With RCP we can achieve both composition and acquaintance: composition when the pointed object is not shared, and acquaintance when it is shared.

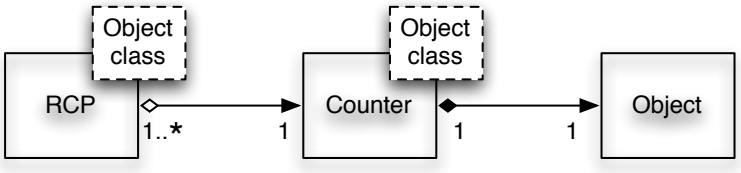


Figure 2.6: RCP mechanism

The use of RCP provides services such as:

- allocation and deallocation of the shared object;
- copy construction;
- *equal to* and *less than* operators, in order to be used in sorting and classifier algorithms;
- assignment operator.

A simplified RCP implementation is reported in Code Listing 2.1.

2.7 Object Configuration

2.7.1 Problem statement

Each algorithm can have some parameters that can be set according to the end-user choices. In a complex object oriented program, these algorithms are widespread inside many different objects. Hence we set an algorithm's parameter through the options of its encapsulating object. However, the end-user has no direct access to the encapsulating objects, but only to the program's main function. The main function is the door through which the end-user can feed algorithms' objects with his/her parameter values. What may happen is that, starting from the objects instantiated in the main function, the parameters we want to set will be passed from one object to another, following the complex chain of relationship between them, until they arrive to the final object encapsulating the algorithm for which the parameters have been set.
This is the natural procedure to set an algorithm's parameter, but it has a troublesome drawback: in order to add an option to a particular object we will need to rewrite all the object's interfaces which link it to the main function. It is clear enough that this is not a practical approach to solving the problem.

2.7.2 Proposed solution

We propose a way to directly pass any type of parameter from the main function to the objects (let's call it *applicant*) that requires it. Our solution consists in letting the applicant be derived from a *Configurator* class. The Configurator class stores a **static** member of class *ParameterList*. A static member has the feature of being shared between all the instances of the same class. Therefore all possible applicants will share the same ParameterList object. Conversely a ParameterList is an STL map that has a *string* as its key, and an *Any* object as its value. *String* stores the name of the parameter, while *Any* is an instance of the *boost::any* class [2], that can store any type of variable. In fig: 2.7 is shown the mechanism to configure the objects.
The user can pass the parameters to Hydra writing an **XML** file. The main function will parse this file storing its data in the ParameterList object. We chose to use an XML format for the input file for some reasons:

- In order not to make the user learn a file format exclusive of Hydra.
- XML emphasizes simplicity, generality, and usability.
- In XML we can easily pass hierachical groups of parameters.
- There are many open source *graphical user interface* (**GUI**) that can be adapted to compile XML files, so in case an Hydra user wanted to plug-in its personal GUI, he/she could do it easily.

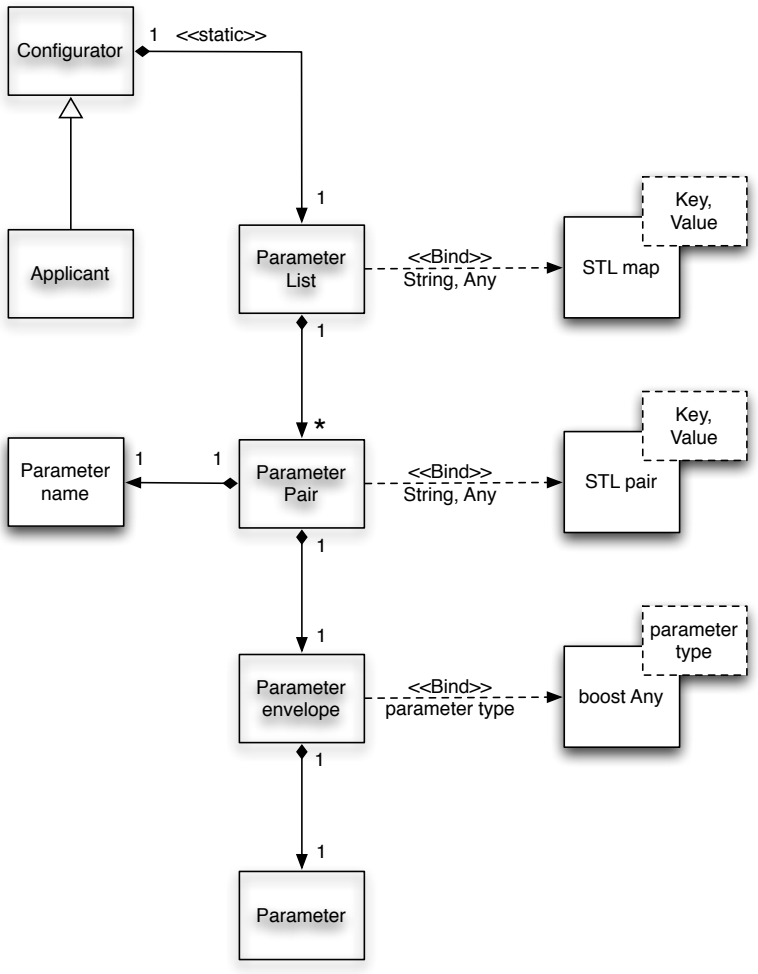


Figure 2.7: Object configuration mechanism

Listing 2.1: Reference Counted Pointer

```
template<class X> class RCP {
public:
  typedef X element_type;
  // Constructor
  explicit RCP(X* p = 0): itsCounter(0) {
  if(p) itsCounter = new counter(p);}
  // Destructor
  ~RCP(){ free();}
  // Copy Constructor
  RCP(const RCP& r) throw() {grab(r.itsCounter);}
  // Overload Assignment
  RCP& operator=(const RCP& r){
    if (this != &r) {
      free();
      grab(r.itsCounter);
    }
    return *this;
  }
  // Overload equal operator
  bool operator == (const RCP& r) const
    {return ( itsCounter->ptr == r.get() );}
  // Overload less operator
  bool operator < (const RCP& r) const
    {return ( itsCounter->ptr < r.get() );}
  // Get object pointed to
  X& operator*() const throw(){return *itsCounter->ptr;}
  X* operator->() const throw(){return itsCounter->ptr;}
  X* get() const throw(){return itsCounter ? itsCounter->ptr : 0;}
  bool unique() const throw()
    {return (itsCounter ? itsCounter->count == 1 : true);}

private:
  struct counter{
    counter(X* p = 0, unsigned c = 1) : ptr(p), count(c) {}
     X* ptr;
    unsigned    count;
  }* itsCounter;
  void grab(counter* c) throw(){ // increment the count
    itsCounter = c;
    if (c) ++c->count;
  }
  void free() { // decrement the count, delete if it is 0
    if (itsCounter){
      if (--itsCounter->count == 0) {
        delete itsCounter->ptr;
        delete itsCounter;
      }
      itsCounter = 0;
    }
  }
};
```

Programs = Algorithms + Data Structures.

Niklaus Wirth

Chapter 3

Data Structure

3.1 Context

In this chapter we will describe the development and implementation of our flexible parallel data structure for scientific computing (**SCDS**). Scientific Computing is dominated by mesh-based analysis methods (**MBAM**s), like finite element method (FEM) and finite volume method (FVM). In a MBAM there are four main actors:

- *The Geometric Model*, which houses the geometric definition of the computational domain.
- *The Mesh*, which describes the discretized representation of the computational domain.
- *The Fields*, which represent the distribution of the independent variables over the mesh entities.
- *The Attribute,* which accounts for the rest of information needed to define and solve the problem.

A SCDS needs to store all these data. In the attribute definition we consider also all mesh-related data, that is, all information associated with particular mesh entities. Application's defined data (**ADD**) and geometric model-to-mesh relationships are examples of mesh-related data.
ADD can come out, for example, from the need of an algorithm to store mesh-based variables whose computation and use takes place at different times. If the cell entity with which the variables are associated, migrates to another process between the time of computation and the time of use, then the variables shall have to migrate to the new process as well. This means that it is advantageous to allow the SCDS to store also ADD. In such a way the applications needn't concern themselves with parallel issues. Besides, in order to have an optimized and cleaner (this feature has not to be underestimated) code, it is always better to have a single SCDS to store all mesh-related data. The computational domain, as built by a CAD, represents the geometry of the problem; let's call it the *geometric domain* (**GD**): Ω_G. The space occupied by the elements which discretize the GD constitute the *mesh domain* (**MD**): Ω_M. Normally it is $\Omega_M \subseteq \Omega_G$, this is because when the geometry's boundaries are curved, then the elements' faces rarely happen to lie on them. This is obviously true for planar faces and straight edges,

but it is often true even for higher order elements. There are actually several procedures, such as mesh refinement, which need to take into account the relation betwen the geometric model and the mesh, therefore it is crucial to have a representional scheme that can reliably represent this relationship.
In conclusion we want to be able to establish a biunivocal association between arbitrary data and each mesh entity.
In the following paragraphs we shall attempt the description of our SCDS. Each paragraph is devoted to one facet of our SCDS. The facets we have individuated are itemized below:

1. Type of mesh representation.
2. Data structure implemenation.
3. Management of parallelism.
4. Data input and output.
5. Interfacing of the SCDS with an application.
6. Computation of the geometric variables.

3.2 Topology-Based Mesh Data Structure

For the *mesh representation* (**MR**) we have decided to rely on a Topological-based data structure [26], that is, on the explicit representation of topological entities and their hierarchy. This choice, together with OOP allows us to:

- Easily store the attributes of an entity inside its object.
- Transparently query for attributes and connectivity between entities.

There are three functional requirements for the design of a general topology-based mesh data structure: topological entities, geometric classification, and adjacencies between entities.

3.2.1 Topological entities

First of all let's introduce some notation:

- $\{X\}$ denotes a given set of Xs.
- $Y_i\{X\}$ denotes the set of all entities of type X belonging to the entity i of type Y.
- $\{Y\{X\}\}$ denotes a set of sets: a given set of Y entities, each of which is defined by a set of X entities.

Topology provides an unambiguous, shape-independent abstraction of the mesh. The representation of general, even non-manifold, geometric objects is complicated, as it requires the use of *loop* and *shell* entities. Fortunately in the case of meshes we can assume some ad-hoc restrictions on the topology which allow the representation of a mesh simply as a set of 0 to

d dimensional topological entities M, where d is the dimension of the computational domain. The full set of mesh entities in 3D is:

$$\{\{M^0\},\{M^1\},\{M^2\},\{M^3\}\} \tag{3.1}$$

where $\{M^d\}$, $d = 0, 1, 2, 3$, are, respectively, the set of vertices, edges, faces, and cells. Each topological entity M_i^d is bound by a set of topological entities of dimension $d-1$: $M_i^d\{M^{d-1}\}$, where $M_i^d\{M^{d-1}\}$ is the set, perhaps ordered, of all entities of dimension $d-1$ belonging to the mesh entity i of dimension d.
The restrictions on the topology which allow this simple representation are:

1. Cells and faces have no interior holes.
2. Each entity of order d, M_i^d, may use a particular entity of lower order, p, M_j^p, with $p < d$, at most once.
3. For any entity M_i^d, there is a single set of entities of order $d-1$, $M_i^d\{M^{d-1}\}$ that are on the boundary of M_i^d.

The first condition allows cells to be represented by one shell of faces that binds them, and faces to be represented by one loop of edges that binds them. The second restriction allows us to define the orientation of an entity based on its boundary entities without the introduction of auxiliary entities. For example, the orientation of an edge M_i^1 bound by vertices M_k^0 and M_j^0, with $k \neq j$, is univocally defined as going from M_k^0 to M_j^0. The third condition means that an interior entity is univocally specified by its binding entities.

3.2.2 Classification

We call *classification* the relationship between the mesh entities and the geometric domain. As we already pointed out, these relationship are fundamental for mesh creation and mesh refinement. For example, when we split a boundary cell into two new ones, we need their boundary faces to follow the geometric boundary in more accurately way than the parent cell's face did. So, in order to make this cell-splitting, we need to keep track of the link between a boundary face M_i^2 and the patch entity G_j^2, that describes the geometry in proximity. If d_i is the dimension of the boundary face and d_j is the dimension of the geometric patch, then: $d_i \leq d_j$ is verified.

3.2.3 Adjacencies

Adjacencies describe how mesh entities are connected to each other. For an entity of dimension d, *first-order adjacency* returns all mesh entities of dimension q, which are on the closure of the entity for a downward adjacency, $d > q$, or for which the entity is part of the closure for an upward adjacency, $d < q$. For denoting specific downward first-order adjacent entity, $M_i^d\{M^q\}_j$, the ordering conventions can be used to enforce the order.
In fig. 3.1 all twelve possible adjacency relationships are shown.

For an entity of dimension d, *second-order adjacencies* describe all mesh entities of dimension q that share any adjacent entities of dimension b, where $d \neq b$ and $b \neq q$. Second-order

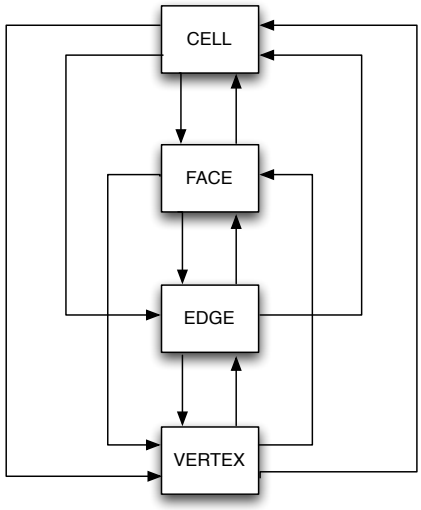
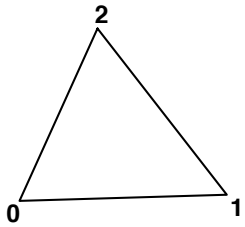


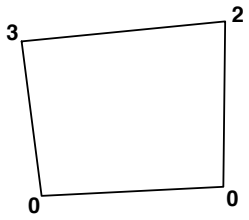
Figure 3.1: 12 adjacencies possible in the mesh representation



Edge	Vertices
1	0,1
2	1,2
3	2,0

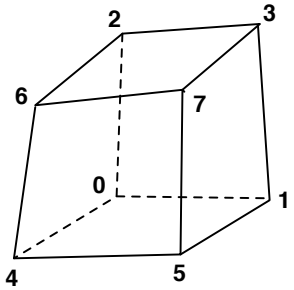
Figure 3.2: Triangle entities numbering convention

adjacencies can be derived from first-order adjacencies. If we take a given face as an example, some adjacency requests that could be made would be: the cells on either side of the face (first-order upward); the vertices binding the face (first-order downward); or the faces that share any vertex with a given face (second-order).
The ordering inside each adjacency set is fundamental in order to perform first-order, second-order and other types of queries explained in paragraph 3.2.5. Therefore it is mandatory to define entity-numbering conventions. In fig.s 3.4, 3.5, 3.6, 3.7, 3.2 and 3.3 are shown our entity numbering conventions.



Edge	Vertices
1	0,1
2	1,2
3	2,3
4	3,0

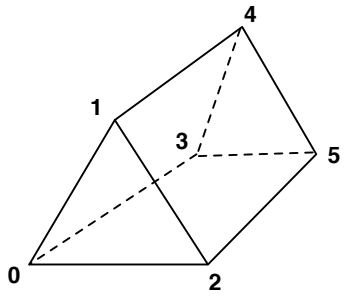
Figure 3.3: Quadrilateral entities numbering convention



Edge	Vertices
1	0,4
2	0,1
3	1,5
4	4,5
5	1,3
6	3,7
7	5,7
8	2,3
9	2,6
10	6,7
11	0,2
12	4,6

Face	Vertices
1	0,1,5,4
2	1,3,7,5
3	3,2,6,7
4	2,0,4,6
5	1,0,2,3
6	4,5,7,6

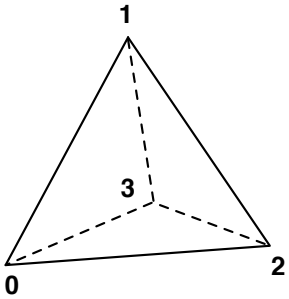
Figure 3.4: Hexahedron entities numbering convention



Edge	Vertices
1	0,1
2	1,2
3	2,0
4	3,4
5	4,5
6	5,3
7	0,3
8	1,4
9	2,5

Face	Vertices
1	0,1,4,3
2	1,2,5,4
3	2,0,3,5
4	0,2,1
5	3,4,5

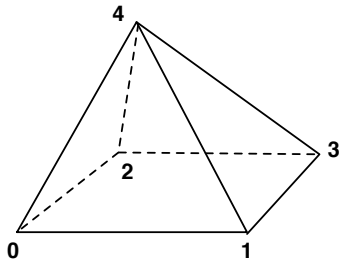
Figure 3.5: Wedge entities numbering convention



Edge	Vertices
1	0,1
2	1,2
3	2,0
4	0,3
5	1,3
6	2,3

Face	Vertices
1	1,0,2
2	0,1,3
3	1,2,3
4	2,0,3

Figure 3.6: Tetrahedron entities numbering convention



Edge	Vertices
1	0,1
2	1,3
3	3,2
4	2,0
5	0,4
6	1,4
7	3,4
8	2,4

Face	Vertices
1	0,2,3,1
2	0,1,4
3	1,3,4
4	3,2,4
5	2,0,4

Figure 3.7: Pyramid entities numbering convention

3.2.4 Mesh representation options

An MR set can be categorized according to two criteria:

- Full or reduced.
- Complete or incomplete.

An MR is full if it stores all 0-to-d-level entities explicitly; otherwise, it is a reduced representation. An MR is complete if it can provide any type of adjacencies requested without involving operations dependent on mesh size. Regardless of whether it is full or reduced, if all adjacency informations are obtainable in $\mathcal{O}(1)$ time, the representation is complete, otherwise it is incomplete.
Following [44], fig. 3.8 shows some statistics and the number of connections for all twelve possible ajacency relationship for tetrahedral and hexahedral meshes.

For pyramidal, wedge-composed, and mixed 3D meshes the statistics and connection numbers are somewhere in between the ones of tetrahedral and hexahedral meshes. Representing all possible adjacency relationship can be quite memory-consuming. The implementation of a mesh data structure must consider the trade-off between storage space required and the time needed to access various adjacency informations. Clearly if more adjacency relationship are stored, less work is required to obtain the adjacencies, but the storage memory required will be greater. The question, then, is determining which set of adjacencies should be stored in order to maximize implementation efficiency. The answer is application dependent. There are applications whose needs require a certain set of adjacencies, whereas other applications may require a completely different set.
Let's make an **example**. A first-order finite volume application for the Euler Equation computes the residuals making two loops:

1. Loop over faces to compute flux contributions.
2. Loop over cells to compute source term contributions.

Considering the first loop, for each face we need to know the conservative variable values of the two cells that share the face, therefore we need our mesh data structure to store the face-to-cell connectivity. Moreover in order to compute volumes, areas, face normal vectors and other geometric parameters, we need our mesh data structure to store also cell-to-vertex and face-to-vertex connectivities. There is no need to store edges. Alternatively, if we want to

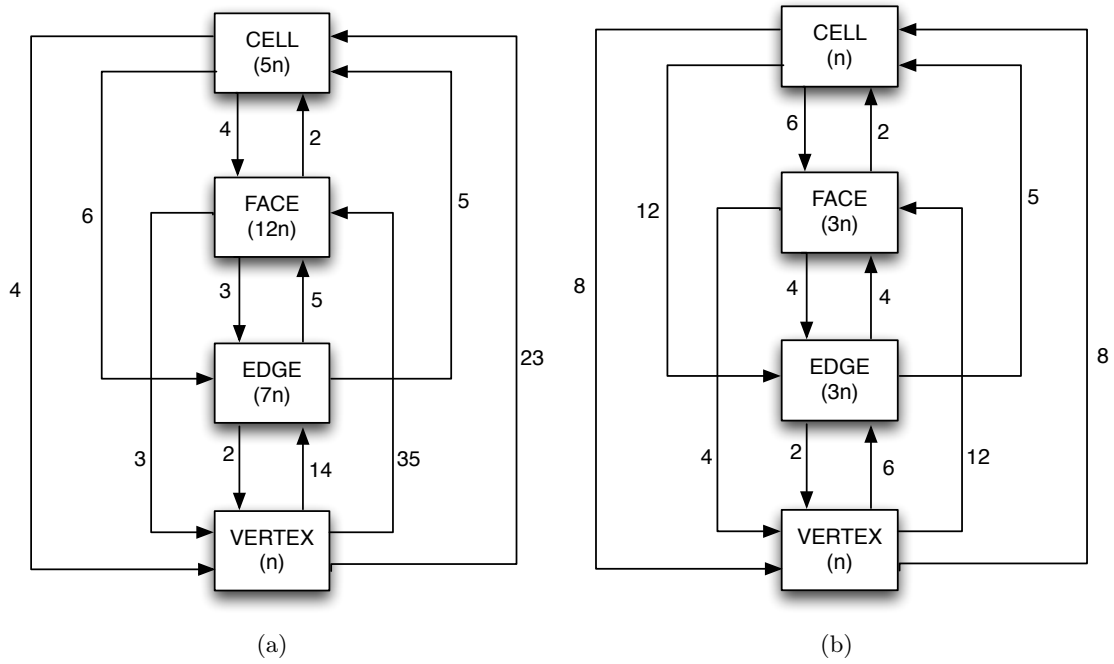


Figure 3.8: Representations of Tetrahedral mesh (a), and Hexahedral mesh (b). Inside the boxes are shown typical statistics for the number of entities. Next to the arrows there are the number of adjacency connections. The numbers for downward adjacencies are exact while the ones for upward adjacencies are averages.

solve the Navier-Stokes Equation with second-order finite volume we may also need cell-to-face connectivities in order to compute variable gradients.
Should we want to apply mesh movement techniques in order to tackle moving boundary problems, we may also need to use the edge-to-vertex connectivity. This is because the algorithm for mesh movement needs to know the edge's length in order to be sure that the new mesh does not have undesired gradings. In fig. 3.9 are shown some examples of MRs.

3.2.5 Flexible mesh representation

There are three main approaches for the design of a mesh data structure:

1. Ad-hoc mesh data structure, shaped for a specific application.
2. Fixed general MR.
3. Flexible MR that is able to shape its representation dynamically based on the needs of the applications.

Since we want to develop a module that shall be useful for every kind of application, we cannot rely on the first approach. The second approach can be the optimum for a certain set of applications but, like all general methods, it may be inefficient for others. Therefore we have decided to develop our mesh data structure following the third approach. Below it is explained

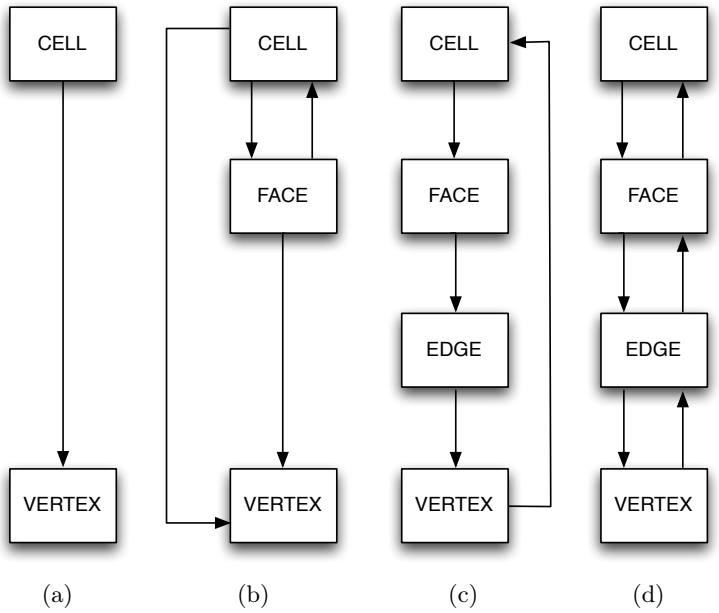


Figure 3.9: Example of 3D mesh representations. Representation (a) is the classic one for FEM, while (b) is suitable for FVM. Representation (c) and (d) are other 3D representations common in litterature.

how we do this.
From the mesh generator we suppose to receive a mesh representation composed of:

- The list of cell entities: $\{M^3\}$;
- The list of vertex entities: $\{M^0\}$;
- The list of adjacencies $\{M^3\{M^0\}\}$.

We shall denominate this set *minimum required representation* (**MRR**). From a MRR we should be able to build any other type of representation. In order to attain this, we implemented a suitable set of functions which take a particular MR as input, and return a representation for entities and/or adjacencies not yet represented in the given MR. We have grouped these functions in four classes:

1. *Entity set extraction functions.*
2. *Entity creation functions.*
3. *Adjacency inversion functions.*
4. *First-order adjacency query function.*

Any other type of function can be obtained suitably combining the above groups.

Entity set extraction functions

These functions are all $\mathcal{O}(1)$. Given a particular adjacency $M_i^p\{M^q\}$ of entity M_i^p, and using the entity numbering conventions (see paragraph 3.2.3), they can extract a set of entities of dimension q, $\{M^q\}_j$ which defines an entity of dimension r, M_j^r, where $p > r > q$. We implemented the following *set extraction functions*:

1. Given: M_i^3, $\{M^1\}$ and $M_i^3\{M^1\}$; return: $M_j^2\{M^1\}$.
2. Given: M_i^3, $\{M^0\}$ and $M_i^3\{M^0\}$; return: $M_j^2\{M^0\}$.
3. Given: M_i^3, $\{M^0\}$ and $M_i^3\{M^0\}$; return: $M_j^1\{M^0\}$.
4. Given: M_i^2, $\{M^0\}$ and $M_i^2\{M^0\}$; return: $M_j^1\{M^0\}$.

Entity set extraction functions are the bricks for building all other types of functions.

Entity creation functions

Given an MR with representations for $\{M^p\}$, $\{M^0\}$ and $\{M^p\{M^0\}\}$ they return $\{M^q\}$, $\{M^q\{M^0\}\}$ (where $p > q$), and eventually $\{M^q\{M^p\}\}$ and $\{M^p\{M^q\}\}$. These functions work making a loop over $\{M^p\}$ and thanks to the *set extraction functions* they extract from each M_i^p the set $M_i^p\{M^q\{M^0\}\}$. Each element $M_k^q\{M^0\}$ of the set $M_i^p\{M^q\{M^0\}\}$ can define an entity M_j^q of $\{M^q\}$ (see paragraph 3.2.1). Therefore these functions check inside $\{M^q\}$ if an entity defined with $M_k^q\{M^0\}$ already exists. If this is the case, the functions may eventually insert M_j^q into the adjacency $M_i^p\{M^q\}$, otherwise the functions create the new entity M_j^q, defined by $M_k^q\{M^0\}$, and insert it both in $\{M^q\}$ and, eventually, in $M_i^p\{M^q\}$. Moreover they also eventually insert M_i^p in $M_j^q\{M^p\}$. From this explanation we can understand that these functions are all $\mathcal{O}(n^2)$. Here is the list of those we implemented:

1. Given: $\{M^3\}$, $\{M^0\}$ and $\{M^3\{M^0\}\}$; return: $\{M^2\}$, $\{M^3\{M^2\}\}$ and $\{M^2\{M^0\}\}$.
2. Given: $\{M^3\}$, $\{M^0\}$ and $\{M^3\{M^0\}\}$; return: $\{M^2\}$, $\{M^3\{M^2\}\}$, $\{M^2\{M^3\}\}$, and $\{M^2\{M^0\}\}$.
3. Given: $\{M^3\}$, $\{M^0\}$ and $\{M^3\{M^0\}\}$; return: $\{M^2\}$, $\{M^2\{M^3\}\}$, and $\{M^2\{M^0\}\}$.
4. Given: $\{M^3\}$, $\{M^0\}$ and $\{M^3\{M^0\}\}$; return: $\{M^1\}$, $\{M^3\{M^1\}\}$, and $\{M^1\{M^0\}\}$.
5. Given: $\{M^3\}$, $\{M^0\}$ and $\{M^3\{M^0\}\}$; return: $\{M^1\}$, $\{M^3\{M^1\}\}$, $\{M^1\{M^3\}\}$, and $\{M^1\{M^0\}\}$.
6. Given: $\{M^3\}$, $\{M^0\}$ and $\{M^3\{M^0\}\}$; return: $\{M^1\}$, $\{M^1\{M^3\}\}$, and $\{M^1\{M^0\}\}$.
7. Given: $\{M^2\}$, $\{M^0\}$ and $\{M^2\{M^0\}\}$; return: $\{M^1\}$, $\{M^2\{M^1\}\}$, and $\{M^1\{M^0\}\}$.
8. Given: $\{M^2\}$, $\{M^0\}$ and $\{M^2\{M^0\}\}$; return: $\{M^1\}$, $\{M^2\{M^1\}\}$, $\{M^1\{M^2\}\}$, and $\{M^1\{M^0\}\}$.
9. Given: $\{M^2\}$, $\{M^0\}$ and $\{M^2\{M^0\}\}$; return: $\{M^1\}$, $\{M^1\{M^2\}\}$, and $\{M^1\{M^0\}\}$.

Adjacency inversion functions

Given an MR with representations for $\{M^p\}$, $\{M^q\}$ and $\{M^p\{M^q\}\}$ with $p > q$, they create a representation for $\{M^q\{M^p\}\}$. These functions are all $\mathcal{O}(n)$. Here is the list of those we implemented:

1. Given: $\{M^3\}$, $\{M^2\}$ and $\{M^3\{M^2\}\}$; return: $\{M^2\{M^3\}\}$.
2. Given: $\{M^3\}$, $\{M^1\}$ and $\{M^3\{M^1\}\}$; return: $\{M^1\{M^3\}\}$.
3. Given: $\{M^3\}$, $\{M^0\}$ and $\{M^3\{M^0\}\}$; return: $\{M^0\{M^3\}\}$.
4. Given: $\{M^2\}$, $\{M^1\}$ and $\{M^2\{M^1\}\}$; return: $\{M^1\{M^2\}\}$.
5. Given: $\{M^2\}$, $\{M^0\}$ and $\{M^2\{M^0\}\}$; return: $\{M^0\{M^2\}\}$.
6. Given: $\{M^1\}$, $\{M^0\}$ and $\{M^1\{M^0\}\}$; return: $\{M^0\{M^1\}\}$.
7. Given: $\{M^0\}$, $\{M^1\}$ and $\{M^0\{M^1\}\}$; return: $\{M^1\{M^0\}\}$.

First-order adjacency query function

These function are all $\mathcal{O}(1)$. We implemented the following set of them:

1. Given: M_i^3, $\{M^2\}$, $\{M^1\}$, $M_i^3\{M^2\}$ and $\{M^2\{M^1\}\}$; return: $M_i^3\{M^1\}$.
2. Given: M_i^3, $\{M^2\}$, $\{M^0\}$, $M_i^3\{M^2\}$ and $\{M^2\{M^0\}\}$; return: $M_i^3\{M^0\}$.
3. Given: M_i^3, $\{M^1\}$, $\{M^0\}$, $M_i^3\{M^1\}$ and $\{M^1\{M^0\}\}$; return: $M_i^3\{M^0\}$.
4. Given: M_i^2, $\{M^1\}$, $\{M^0\}$, $M_i^2\{M^1\}$ and $\{M^1\{M^0\}\}$; return: $M_i^2\{M^0\}$.
5. Given: M_i^0, $\{M^1\}$, $\{M^2\}$, $M_i^0\{M^1\}$ and $\{M^1\{M^2\}\}$; return: $M_i^0\{M^2\}$.
6. Given: M_i^0, $\{M^1\}$, $\{M^3\}$, $M_i^0\{M^1\}$ and $\{M^1\{M^3\}\}$; return: $M_i^0\{M^3\}$.
7. Given: M_i^0, $\{M^2\}$, $\{M^3\}$, $M_i^0\{M^2\}$ and $\{M^2\{M^3\}\}$; return: $M_i^0\{M^3\}$.
8. Given: M_i^1, $\{M^2\}$, $\{M^3\}$, $M_i^1\{M^2\}$ and $\{M^2\{M^3\}\}$; return: $M_i^1\{M^3\}$.

3.3 Data Structure Implementation

3.3.1 Mesh and field representation

Domain decomposition

We program in parallel supposing that a *Distributed-Memory* **MIMD** environment is used. This means we suppose that each physical process has its own private memory. In opposition to distributed-memory, there is the paradigm of shared-memory, where the physical processes share the same memory. A program tailored for a distributed-memory environment is more general than one for designed for shared-memory, as the former can also run in a shared-memory

environment, but not viceversa. The parallel programming paradigm used in our case is: *single-program multiple-data* (**SPMD**). This means that each process runs the same program, but over different data. When we need to run a program only over particular processes we use branch statements. The natural way to apply SPMD for mesh-based simulations is to divide the mesh into non-overlapping sub-meshes, and assign each sub-mesh to a single process. We call these sub-meshes *partition domains*. In this way each process is responsible for the driving of the computation over its partition domain: we express this concept saying that a process has *ownership* of the entities of its partition domain.
It should appear obvious that the computation over a certain partition domain cannot be performed if the process that owns it does not have information regarding what happens all around it as well. Thus a process also needs to store information concerning the ribbon of mesh immediately adjacent to its partition domain. Clearly, this ribbon is part of other partition domains, due to which we also call it *overlap region*.
A process must store in its private memory a copy of the overlap region data, it cannot rely on the private memory of the processes owning them. If this were the case, it would mean that each time a process needed access to other processes' data, a *communication* between the two processes would have to be made. Unfortunately, the communication of a variable of a certain type spends quite much more time than performing an arithmetic operation, like addition or division, between two variables of the same type. The cost of a communication, T, expressed in seconds, can be computed as:

$$T = t_s + kt_c \tag{3.2}$$

Where t_s is the communication start-up time, k is the size of the variable to be sent, and t_c is the time to transmit a unit of information. In most systems t_c is within one order of magnitude of the cost of an arithmetic operation, t_a, and t_s is from one to three orders of magnitude greater than t_c. We immediately conclude that we must reduce the number of communications, trying to group them as much as possible.
Summarizing, each process must store two pieces of mesh:

1. Its partition domain; in which it drives the computation.
2. The corresponding overlap region; where the computation is driven by other processes, but whose data are needed to drive the computation of the partition domain.

The storage within each process of both its partition domain and corresponding overlap region has two advantadgeous consequences: *locality* and *transparency*. Locality means that all data necessary for driving the computations in a partition domain must be accessible locally to the process, in such a way that the algorithms need not explicitly fetch data stored in the memory of other physical processors. Transparency means that all data can be accessed in the same way, irrespectively of their origin, that is, the process which owns them. The most relevant outcome of this is that it allows an application programmer not to care about parallel issues: the data storage is transparent with respect to the subdivision of the mesh across processes.

Storage typologies

We have designed our scientific computing data structure, SCDS, to have two basic massive storage typologies:

1. one for the storage of the topological entities composing the mesh named: *distributed mesh container*;
2. and the other for the storage of matrices and vectors involved in the system matrix named: *distributed field container*. One of these vectors is always the one representing the dependent variable of the computation: the *field vector*.

Each topological entity type is implemented with a suitable class that contains information on:

- adjacencies;
- attributes;
- other necessary data.

All matrices and vectors involved in the system matrix are built composing smaller vectors and matrices, each one associated to a single topological entity (*field topological entity vector*, FTEV, and *matrix*, FTEM). We could store FTEVs and FTEMs inside their entity's class, but we find this solution not convenient, because the BLAS library, used to solve the matrix systems, is optimized to solve linear systems in which vectors and matrices occupy a continuous piece of computer memory. Therefore we decided to store FTEVs and FTEMs inside the aforementioned BLAS-friendly system matrix members.
All data storages are allocated as instances of RCPs (see paragraph 2.6), so that we can:

1. Allocate them dynamically. We do not need to statically instantiate them.
2. Pass them all around programs (one can also read: objects) that need them. The storages are passed from one program to another without copying them, and therefore saving a lot of time and memory.
3. Share them between many programs (one can also read: objects). If a program updates or modifies a storage, the changes are immediately and automatically seen by every other program sharing the storage.

We shall now describe these two massive storage types.

3.3.2 Distributed mesh container

Entity representation

Space Region. We assign every entity to a *space region*. A space region is a group of entities that represents a particular part of the computational domain. It can be defined either by the user for his/her purposes, or by an application. Some examples of space regions could be:

- Each set of boundary faces which define a particular boundary condition.
- A set of internal cells within which a particular algorithm has to be applied.
- A set of internal cells which have particular values of certain physical variables.

Each space region is identified by its name, and it is passed to the data structure by the parameter list (see paragraph 2.7). We store the list of space region names in a vector, therefore we can also identify the space region with their name's position inside this vector. This is a rather convenient strategy, as an integer occupies less memory than does a string of characters. A SpaceRegion object takes care of storing data regarding:

- Boundary condition definition, if the SpaceRegion object defines a domain boundary region.
- Attributes which are common to all the entities belonging to the given space region.

Topological Entities. In our implementation we have followed a basic principle of OOP: If some classes have some common data and members, then one should group these common elements in a base class and let the other classes derive from this base one. In fig. 3.10 is shown the UML class diagram of the topological entities.

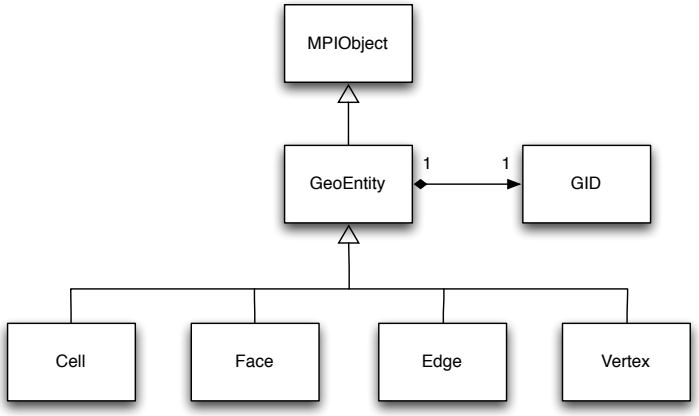


Figure 3.10: UML diagram for the representation of the topological entities.

MPIObject. This class groups MPI information, such as:

- rank of the process currently housing an instance of this class;
- number of processes in the *communicator world* of the process currently housing an instance of this class;

GID. GID is the class which represents the global identifier of a topological entity. It stores two basic kinds of information:

- the entity identifier (EID);
- the rank of the process (PID) currently *owning* the entity object which the GID instance belongs to.

The EID univocally identifies every single entity over all others in all processes. There could be many processes storing a copy of an entity, but there must be only one process that has ownership over the entity. Many methods are implemented inside GID in order to handle the

EIDs and PIDs stored within in whatever situation an entity might be.

GeoEntity. This class is an abstraction of a topological entity. It holds a GID and information regarding:

- If the instance is on the boundary of the domain.
- Which *space region* the instance belongs to.
- The number of physical variables associated with the instance.
- Other flags.

Once more, several methods are implemented here.

Connectivity. This class is a wrap around an STL vector. An STL vector is a dynamic allocatable array template with sophisticated functionalities. An empty STL vector occupies the space for three pointers to *double*. We use STL vectors to store a set of adjacencies. Since the use of a particular adjacency is application-dependent, and since we do not want to waste memory on an empty STL vector, we decided to create a special wrap around it. This wrap instantiates an STL vector only if the adjacency it represents is going to be used.

Cell. This class represent a cell; in fig. 3.11 is shown its simplified UML diagram. Inside some connectivity objects are stored first-order adjacencies with cells, faces, edges and vertices; whereas inside simple STL vectors are stored field and geometric attributes.
From this class, some concrete classes representing specific cell geometries are derived. Currently there are derivations implemented for Hexahedra, Wedges, Pyramids and Tetrahedra, each with nodes coincident with vertices. The derived classes implement methods for computing: volume, central point and other geometric parameters. Inside the Cell parent class are instead implemented functions common to every concrete cell.

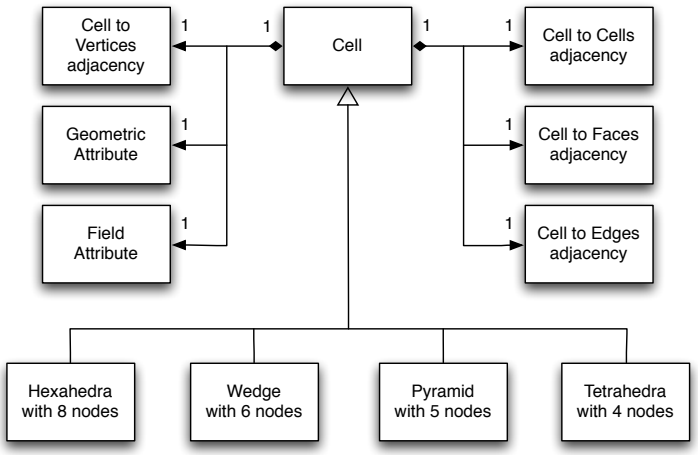


Figure 3.11: UML diagram for the cell class.

Face. This class represents a face; in fig. 3.12 is shown its simplified UML diagram. Inside some connectivity objects are stored first-order adjacencies with cells, faces, edges and vertices; whereas inside simple STL vectors are stored field and geometric attributes.
As in the previously described class, some concrete classes representing specific face geometries are derived from this class. Currently there are derivations implemented for Triangle, and Quadrangle, each with nodes coincident with vertices. The derived classes implement methods for computing parameters such as area, normal vector, central point and other geometric parameters. Inside the Face parent class are instead implemented functions common to every concrete face.

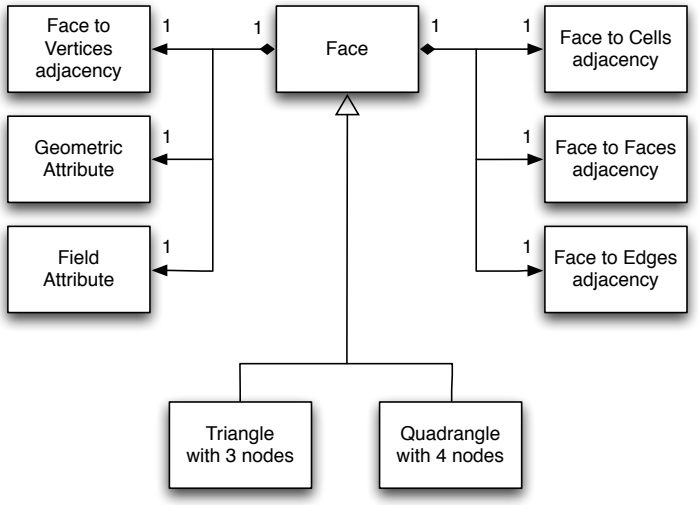


Figure 3.12: UML diagram for the face class.

Edge. This class represents an edge; in fig. 3.13 is shown its simplified UML diagram. Inside some connectivity objects are stored first-order adjacencies with cells, faces, edges and vertices; whereas inside simple STL vectors are stored field and geometric attributes.
As it happened with the Face and Cell classes, from this class is currently derived one concrete class representing an edge with 2 nodes coincident with the vertices. The derived class implement methods for computing length, central point and other geometric parameters.

Vertex. This class represents a vertex; in fig. 3.14 is shown its simplified UML diagram. Inside some connectivity objects are stored first-order adjacencies with cells, faces, edges and vertices; whereas inside simple STL vectors are stored field and geometric attributes.
Vertex maintains also an instance of a Point. Point is a class representing a point in 3D space.

Entity containers

We have found it practical to have eight containers which we can classify depending on whether:

- they store entities belonging to either the partition domain or the overlap region;
- they store cell, face, edge or vertex entities.

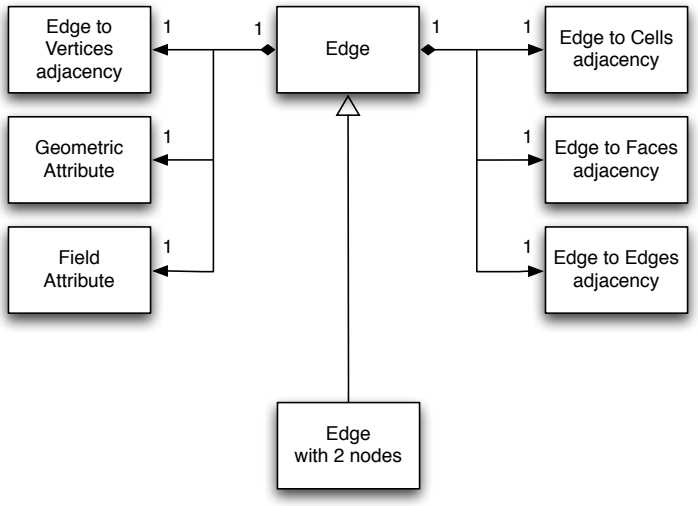


Figure 3.13: UML diagram for the edge class.

These containers are instances of a template container class similar to that of the STL map. In computer science, a map is an associative container composed of a sequence of key-value pairs. It allows for a look-up on the basis of a key. To facilitate quick searching, our implementation of these maps internally ressembles a binary tree. One of its defining characteristic is that elements inserted in a map are sorted on insertion. It also means that, unlike array data structures where elements in a given position can be replaced by others, elements of our map in a given position cannot be replaced by any new elements of a different value. This is because our map would ideally like to have it placed in a possible different location in accordance with its value relative to those in the internal tree.
The entity containers are instances of our map container, where they use an RCP to an entity as value, whereas they use the entity's EID as key.
The reasons why we decided to rely on a map container and not on a classic static array or a dynamic array (like STL vectors) are basically related to the enanched dynamic capability of a map, something which becomes clear when we analyse and compare these three data structures:

Static Array. A static array has clearly no dynamic capabilities. Therefore if we perform h-adaptivity we cannot have new cells added inside it, and we would need to instantiate a new one with the right size. Nevertheless a static array is the most cache-friendly type of container.

Dynamic Array. In a dynamic array we can, in general, add new elements with performance of $\mathcal{O}(1)$ in the best case, or $\mathcal{O}(n)$ in the worst. Instead if we need to delete an element the performance is always $\mathcal{O}(n)$. The need to delete an element might arrise the moment a mesh entity has to change the partition domain it belongs to: it shall be deleted from one partition domain and added to another one.

Map. A map has $\mathcal{O}(1)$ performance both on insertion and deletion. Unfortunately it is the least cache-friendly; but we think this cannot be a irredeemable characteristic, as, for example,

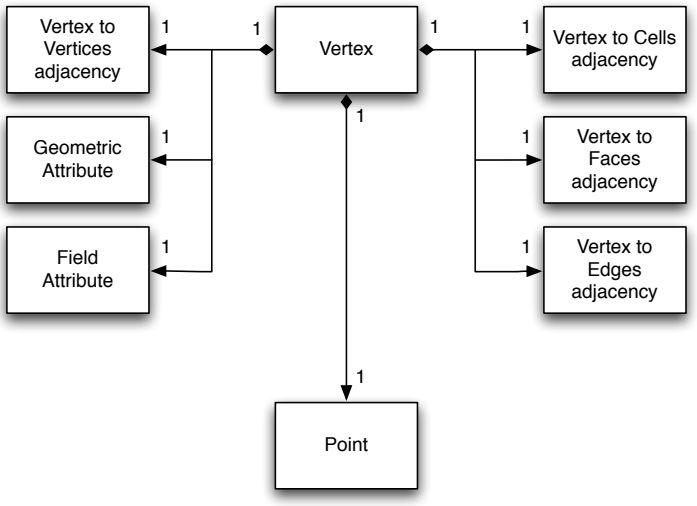


Figure 3.14: UML diagram for the vertex class.

in a loop over entities for the computation of the residuals, the query for a map element happens roughly once per entity. The time used to get a map element is negligible compared to the time needed for the calculation regarding the extracted entity.

Since we want our data structure to be able to deal with dynamic meshes it seems to us that the best compromise is the use of a map data structure for storing mesh entities.

3.3.3 Distributed field container

For the design of the data structure for vectors and matrices involved in the system matrix, we took inspiration from the solution of the Trilinos Library, [58]. This is an advantage when it comes to the task of interfacing Hydra with Trilinos itself as well as with PETSc library for the solution of linear systems. The interface between Hydra and these libraries is absolutely critical: if this task is not permormed correctly, its immediate consequence shall be a slow code.
We defined a class simply called *Vector* which transparently represents a vector across all processes. In the same way we defined a class simply called *Matrix* that transparently represents a matrix across all processes. Both Vector and Matrix store double precision elements. Beside these two classes there is another one called *Map*, responsible for describing the distribution of vector elements and matrix rows across all the processes. The Map class is a wrap around an array of integers which represent the positions, inside Vector and Matrix respectively, of the elements and rows belonging to a process.

3.3.4 Data Holder

We implemented a class, called *IMesh*, that stores both distributed mesh containers and distributed field containers as its members, and works therefore as the data holder. It is in this

class that all functions defined in paragraph 3.2.5 are implemented. It is moreover supplied with many other functions which allow it to operate over the data containers. An instance of IMesh can therefore rightly represent a computational domain.
We can instantiate as many objects of IMesh class as we need. For example, in a fluid-structure coupled simulation, we need to represent two computational domains with the links between them. We can easily do that instantiating two IMesh objects, one for each computational domain. We will come back to these topics in chapter 4.

3.4 Management of Parallelism

3.4.1 Parallel Strategy

One of the targets of manufactory system managers is to maximize the use of resources. Similarly, programmers want to maximize the use of processes. The main way of doing this is to balance the partition domains. Most of the time this means that all partition domains must have the same number of *points*, where, generally, as *points* are intended either the vertices (**node-wise decomposition**) or the cells (**element-wise decomposition**) of a mesh. Other times it is better to make a balance considering weighted points. The weights would be proportional to the computational load associated with each point.
There are no stringent reasons to choose one of element or node wise decomposition. We have decided to stick to element-wise decomposition because it performs better for hybrid meshes, that is, meshes with mixed geometric types of cells. Fig. 3.15 reflects our strategy for the management of parallelism.
The picture illustrates the following steps:

1. Read the mesh from a file, produced by a mesh generator, and load it in one or more processes.
2. Load balance the *graph* of the mesh.
3. Perform the migration of the partition domain:
   (a) mesh data;
   (b) field data.
4. Build the overlap region.
5. Perform the migration of the overlap region:
   (a) mesh data;
   (b) field data.
6. Perform one iteration of the computation.
7. (a) If we have reached the final iteration, or if the computation is not iteration-based, then stop the computation;
   (b) otherwise, check if mesh adaptation is required:

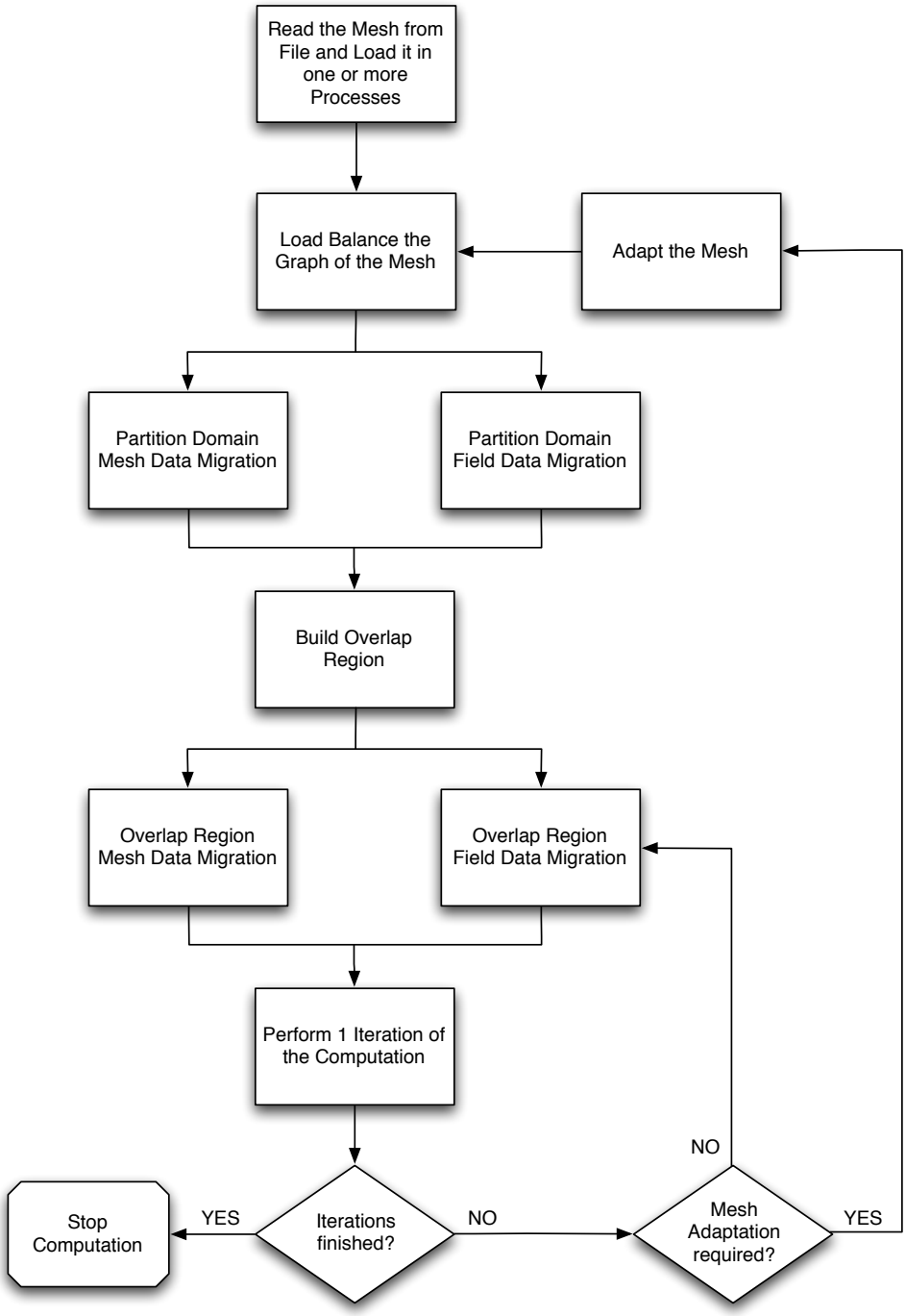


Figure 3.15: Diagram illustrating our parallel strategy.

i. if it is required, then adapt the mesh and go back to step *(3)*;

ii. if it is not required, then just update the overlap region field values going back to step *(5a)*.

3.4.2 Mesh Graph Load Balancing

In order not to reinvent the wheel, the task of load balancing the graph of the mesh is partially accomplished by two external open source libraries: Zoltan [15] and ParMetis [54]. The interface to these libraries is hidden in a *GraphPartition* object, which apart of calling these libraries, it also supplies several partition functionalities. With these two libraries we are able to apply several Graph Partitioning Algorithms:

- Geometric (coordinate-based) Partitioning Algorithms:
  - Recursive Coordinate Bisection;
  - Recursive Inertial Bisection;
  - Space Filling Curve Partitioning.
- Combinatorially (topology-based) Partitioning Algorithms:
  - Graph partitioning;
  - Hypergraph partitioning.

In fact, no single partition algorithm works best for all applications. Trade-offs are based on: Quality vs. speed; Geometric locality vs. data dependencies; and High-data movement costs vs. tolerance for remapping.
Geometric Partitioning Algorithms. They assign objects which are physically close to the same process. They have the advantages of being fast, inexpensive and not requiring connectivity information (the mesh graph). On the contrary, they have the disadvantages of producing mediocre partition quality, needing coordinate information, and they can generate disconnected subdomains for complex geometries.
Combinatorially Partitioning Algorithms. They truly partition the graph of the mesh. A graph is defined by a set of nodes and their connectivities. For a mesh graph the nodes can represent either the vertices or the cells, depending on whether we want to perform node-wise or element-wise decomposition respectively. The advantages of these algorithms are: high quality partition, without disconnected subdomains, and a better control of inter-processes communications. Their disadvantage consists in a larger computational cost.
The GraphPartition object takes an RCP to the DataHolder as its input, and it gives as outputs the following information:

- Exporting information, consisting of a list of pairs detailing:
  - the GID of a cell that has to be exported to another partition domain;
  - the PID (process rank) of the partition domain process to where a cell has to be exported.
- Importing information, consisting of a list of pairs detailing:

- the GID of the cell that has to be imported from another partition domain;
- the PID (process rank) of the partition domain process from where a cell has to be imported.

These pieces of information are collected in an object we called *CellGraph*. With CellGraph we are able to perform the data migration, as we shall see in paragraph 3.4.4.

3.4.3 Overlap Region Construction

In fig. 3.16 is shown an example of an element-wise decomposition for a triangular 2D mesh. We shall make use of this example to illustrate the construction of the overlap region.
After load balancing and partition domain data migration, each process stores only the data of its own partition domain. They should now acquire information about their adjacent overlap region. As it can be seen from the figure, at this moment, the cells of the overlap region of process A are stored in process B, and viceversa. It is therefore required that process A collects the cells that must be sent to B in order to let B have its overlap region; the same has to be done by process B regarding process A.
If we put it in broader terms, each process has to gather the information about the cells that must be sent to all processes with which it shares a section of its partition domain boundary. Faces, edges and vertices that are on the closure of the cells to be sent must be collected for data migration, too.

The algorithm we develop for building the overlap region takes advantage of the cell-to-cell adjacency information. If this adjacency is not built, then the algorithm will do it by itself, and when it has finished building the overlap region it will delete this adjacency. The strength of our adjacency objects (3.3.2) is that they also keep trace of the connectivities between entities belonging to different partition domains.

1. First of all, each process creates a vector, E, which stores entities of type *Pack*. There is one Pack element for each of the other running process. Pack is a class that serves to store all data intended for a particular process. It contains sets of RCPs to cells, faces, edges and vertices. This topic is explained in more detail in paragraph 3.4.4.
2. A loop over the partition domain cells checks if each cell C_i has neighboring cell C_j belonging to other partition domains. If this is the case, then cell C_i and its faces, edges and vertices are all stored in the $r-th$ Pack element of vector E. In this way the first layer of the overlap region is created.
3. For every other layer of the overlap region, we loop over the cells already stored in E and, using the cell-to-cell adjacency information, we easily build one layer at a time.

3.4.4 Data Migration

Mesh Data Migration

One of our aims is to perform as less inter-process communications as possible, as we explained in paragraph 3.3.1. We thus developed the mesh migration facility so as to be able to perform

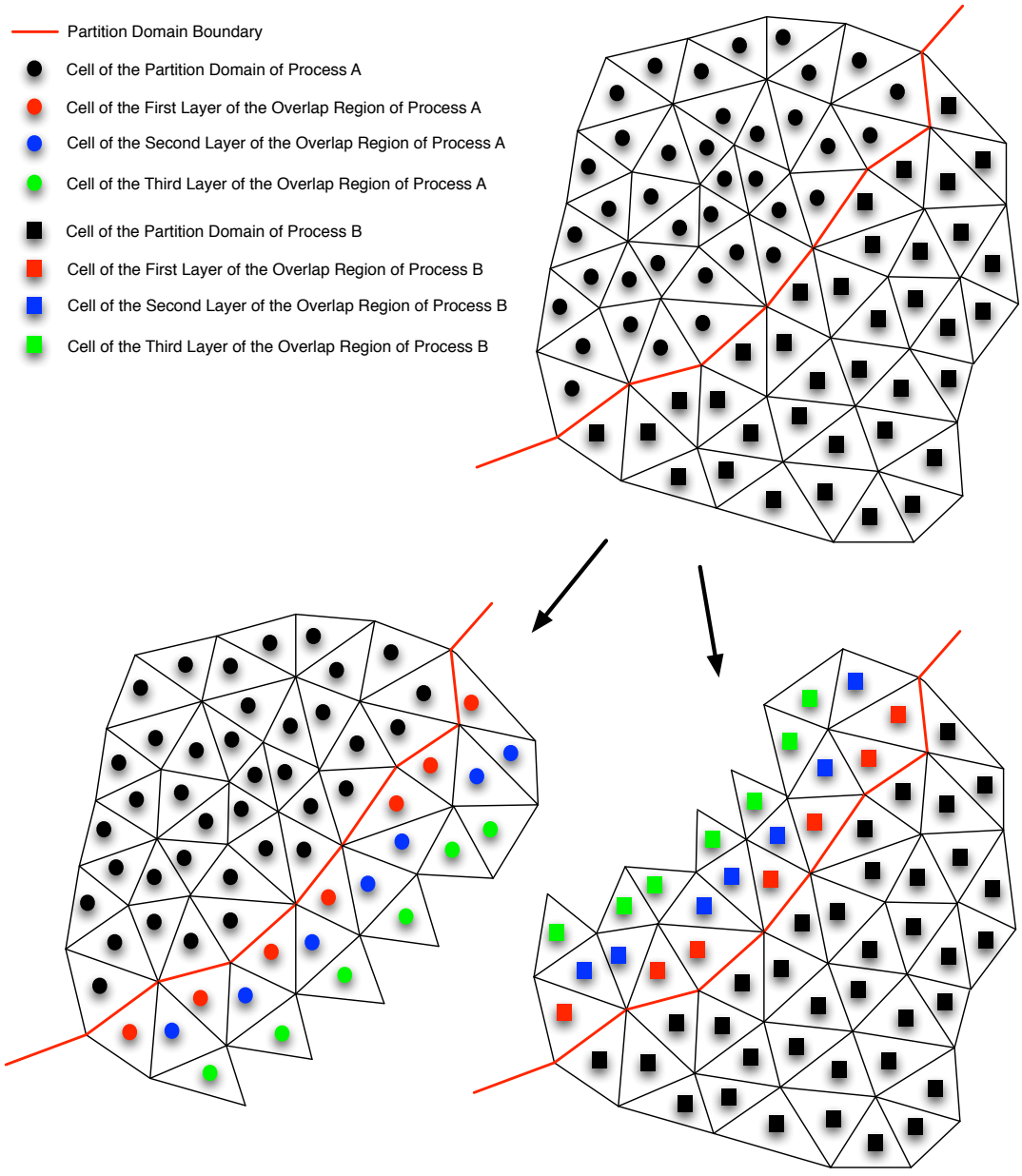


Figure 3.16: Element-wise decomposition.

just one call of an MPI collective communication function per data transmission. We do the communication with a call to *MPI-ALLTOALL*. The communication protocol of MPI-ALLTOALL is explained in fig. 3.17. On the left side of the graph we can see the data distribution across all processes before the communication. Each process stores a group of data for each processor, including itself, we call this set of data groups as *Export data*. On the right side of the figure we can see the situation after the communication has taken place. We see that each process now stores all data that all other procesess, including itself, had prepared for exportation; we call this set of dat groups as *Import data*.
We implemented the Export and Import data as STL vectors of *Pack*s. In fig. 3.17, each square can be considered as a concrete rendition of elements belonging to Pack class. This class can contain either:

- The data that process i must send to process j: exporting data;
- The data that process j is importing from process i: importing data.

A Pack object stores four STL sets of RCPs, one for each topological entity type (cell, face, edge and vertex). An STL set is an abstraction of the mathematical concept of *set*. An STL set makes posssible the quick identification of any of its elements based on the keys attached univocally to each one of them; in our implementation, the keys are the EID linked to every single entity. As an STL set does not allow for duplicates, we can be sure that we are not exporting or importing useless duplicates.

Unfortunately, we can not send an Export object as it is. MPI only allows the transfer of

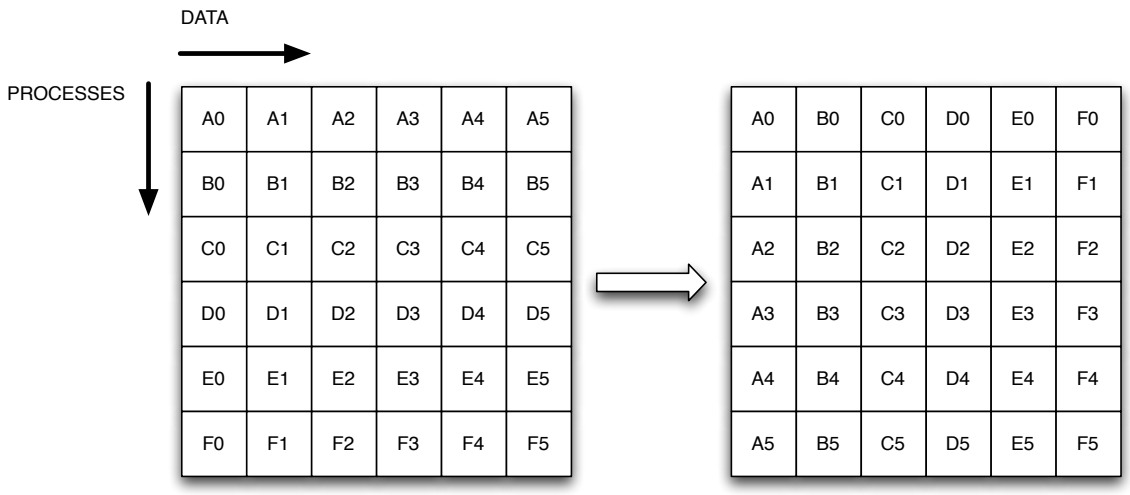


Figure 3.17: MPI ALLTOALL communication protocol. The left and right matrix represent the data distribution, respectivelly, before and after executing the communication.

basic data types, such as: unsigned, integer, float, and double. But Pack objects are not at all basic data types; they are instances of a quite complex class, so they cannot be transferred. To overcome this problem, the standard solution is to build an MPI-DATATYPE.
An MPI-DATATYPE represents the memory scheleton of a complex variable, like a C++ class. Once this scheleton has been built, it must be given the variable's memory starting position. This solution generates fast communication but it leads to a loss of flexibility. In fact MPI-DATATYPE supposes that the variable's memory scheleton is constant, so we cannot consider

dynamically resizable objects. Moreover, in our opinion, programming the memory scheleton of a variable is tiresome and error prone, so the code extensibility is not facilitated. To circumvent the obstacle we decided on a different strategy: *serialization*.
To serialize a C++ object means decontructing it in a sequence of bytes. Bytes are basic types that MPI is able to transmit between processes. Once the sequence of bytes has been transmitted to another process, it shall have to be reshaped into the object they previously represented: the object has to be reconstructed. Each object that must be moved between processes would then need to implement two functions: one that deconstructs it in bytes, and another that reconstructs it in its normal form. What has to be deconstructed and reconstructed in an object are its member data, not its functions. This is because in C++, the functions live in a separate memory area common to all objects belonging to the same class. Therefore, as all the processes are instances of a certain class, then the class' functions will have already been allocated. Member data are instead specific to each individual object.
The Pack class and all the classes that have a composition or inheritance relation with Pack have to implement deconstruction and reconstruction functions; this implies that even the classes of cell, face, edge and vertex have to implemement these two functions.
Our approach could be summarized in the following way: an Export Pack object will be deconstructed, then sent to another process, and finally reconstructed in an Import Pack object. As explained in paragraph 3.4.1 we need to perform two data migrations: one for partition data, and the other for overlap regions. In each case we should build an Export object, but:

- For the partition domain data, Export is built according to the CellGraph object, 3.4.2.
- For the overlap region data, Export is built with the Overlap Region Construction, 3.4.3.

When a process receives partition domain data, it pushes them back into its own partition domain mesh containers. On the other hand, before receiving the new Overlap Region data, a process clears the whole contents of its overlap region mesh containers, and only then it may begin storing the new contents. For the sake of clarity we shall omit here all technicalities involved in these operations.

Field Data Migration

The procedure for field data migration is quite easier, and faster, than that of the mesh. In fact all field are variables are of type *double*, so they can be sent directly by MPI. In paragraph 3.3.3 we have explained that we associate a vector to each field vector and matrix; this vector, called Map, describes the parallel distribution of their elements and rows respectively. When field data (referring either to partition or overlap region) need to migrate, we construct a new Map, which describes the new parallel distribution of matrix rows and vector elements. With these two maps we are able to call MPI-ALLTOALL in order to transfer elements and rows and obtain the new parallel distribution. Again we shall not go into technical details here as to how this is achieved.

3.5 Interfacing the data structure with application programs.

3.5.1 Iterators

The implementation details of our data structure are quite complex. Therefore it is better to develop a tool that allows the application's programmer not to concern himself/herself with concrete details of the data structure. We base this tool on the Iterator design pattern, [42]. An iterator object provides an abstract view of the data structure as a sequence of objects. In our case, these objects should represent cells, faces, edges and vertices, thus we have implemented one Iterator class for each one of them.
We have designed our iterator classes to be a complete interface to the entities they represent. With them we have transparent access to:

- The geometric features of the entity the iterator refers to, such as volume for cell entities, or area for face entities.
- The sub matrix (FTEM) and sub vector (FTEV) parts of the matrices and vectors involved in the system matrix (see paragraph 3.3.1) which are associated to which the iterator refers to.
- All the adjacencies of the entity the iterator refers to.
- All the attributes that an application program has stored inside the entity object the iterator refers to.

Concerning the traversing of the data structure, we have developed two modalities:

1. **Global Traversing.** That is, traversing of the whole set of partition domain cells, faces, edges, vertices as a sequence. In this way we can iterate over all the cells, faces, edges, vertices of a partition domain.
2. **Local Traversing.** That is, traversing of the entities of a partition domain and its related overlap region as a single whole, moving from an entity to another following the adjacency stored in the data structure. For example, suppose that the cell-to-face and face-to-cell adjacencies are defined; thus given an iterator to a cell, we can get an iterator to one of its faces from it. Then from the obtained face iterator we can get an iterator to the other cell that shares it, and so on.

In order to work, an iterator object needs just have acquaintance of the Data Holder. For this purpose we need just to pass an RCP to IMesh in the iterator constructor.
An Iterator object stores internally only:

- An RCP to an IMesh object (the Data Holder).
- The GID of the entity it is currently pointing to.
- A mesh entity container's iterator pointing to the GID entity.

Therefore, it is a very lightweight object, so it is fast to instantiate and it occupies very little memory. In fig. 3.18 is shown the UML diagram of the essential elements an Iterator object.

One of the most notable features of Iterator class is that we can:

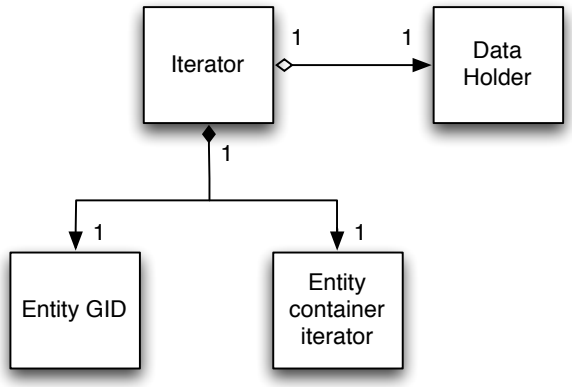


Figure 3.18: Essential Iterator UML diagram.

1. Instantiate as many objects as we want at the same time, also objects referring to the same entity.
2. Work with one or more Iterator objects at the same time, letting them interact with each other.
3. Delete an Iterator object when it is not required anymore.

3.5.2 Data Input and Output

In order to use our data structure, an application program need only instantiate the IMesh class inside an RCP. Then, it sends a request to IMesh in order to build the data structure, the data structure is ready to be used by the application program simple as that.
So as to build the data structure, IMesh has to read a file generated by a mesh generator. At present only *Gambit* mesh files are supported. Gambit is a very famous mesh generator, unfortunatelly it is not currently licensed by its software house anymore. In any case most CFD users still have a working installation of Gambit in their computers. We foresee the implementation of reader functions to other mesh generators' files in the near future.
Once the simulation has been made, the huge set of numbers produced is saved in a VTK file, [14]. In this way we can visualize the solution in any visualization software that supports this type of file format. Paraview [10] is a very famous open source software for scientific visualization that natively supports VTK file format. In the near future we intend to implement a file writer which shall provide data file format to be read with Tecplot [12], the most extensively used commercial scientific visualization software nowadays, as well.

3.6 Geometric variables

In this paragraph we explain how we compute some geometric parameters associated to arbitrarily-shaped faces and cells. This is a topic that should not be underestimated, because an unsuitable

computation of these parameters can, curiously, slow down the computation, or bring about inexact solutions.

3.6.1 Face area

For the following explanation we refer to fig. 3.19

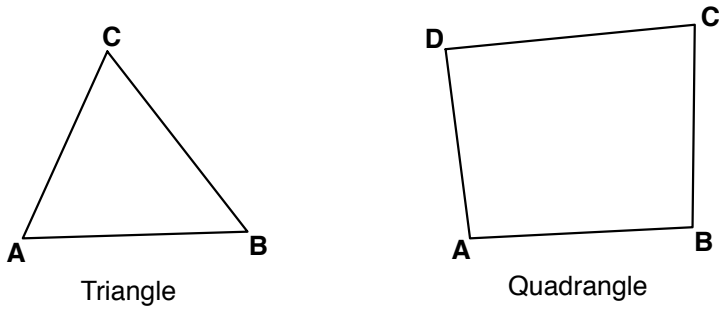


Figure 3.19: Definition of the face vertices.

For a triangular face of vertices A, B, C we compute its normal vector as:

$$\bar{S}_{CDA} = \frac{1}{2}(\bar{X}_{AC} \times \bar{X}_{CD}) \tag{3.3}$$

For a quadrilateral of vertices A, B, C, D we can compute its normal vector as:

$$\bar{S}_{CDA} = \frac{1}{2}(\bar{X}_{AC} \times \bar{X}_{BD}) \tag{3.4}$$

Equation 3.4 is applicable even when the four vertices of the quadrilateral are not coplanar. The area, A, of a face can be computed as the module of the face's normal vector:

$$A = \|\bar{S}\|_2 \tag{3.5}$$

3.6.2 Face normals

Unit face normal vector can be easily computed from:

$$\bar{n} = \frac{\bar{S}}{A} \tag{3.6}$$

3.6.3 Cell volume

For the following explanation we refer to fig. 3.20

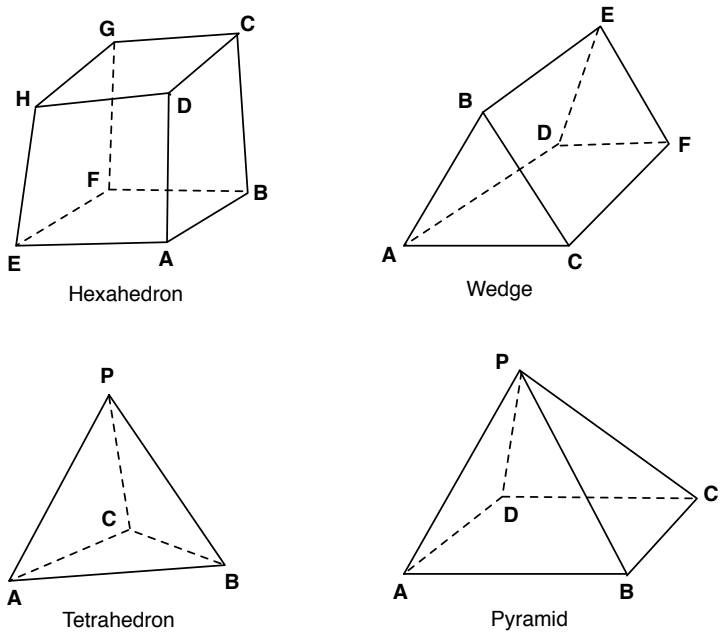


Figure 3.20: Definition of the cell's vertices.

Volume of a Tetrahedron

The Gauss divergence theorem states that the outward flux of a vector field through a closed surface is equal to the volume integral of the divergence of the region inside the surface:

$$\int_{\Omega} (\bar{\nabla}\bar{a}) d\Omega = \oint_{S} \bar{a} d\bar{S} \tag{3.7}$$

Where Ω is an arbitrary volume, S is the closed boundary surface, and a is a scalar variable. Taking vector $\bar{a}$ equal to the position vector $\bar{x}$ we obtain the general formula for computing the volume, Ω, of an arbitrarily shaped cell:

$$\Omega = \frac{1}{3} \oint_{S} \bar{x} \cdot d\bar{S} \tag{3.8}$$

Applying this formula to a tetrahedron of vertices $PABC$, we obtain:

$$\Omega_{PABC} = \frac{1}{3} \sum_{faces} \bar{x} \cdot \bar{S}_{faces} \tag{3.9}$$

In equation 3.9 the vector $\bar{x}$ has its end point in the corresponding face. If we consider the start point in one of the vertices of the tetrahedron, say P, then the scalar product is null for all faces save for ABC, so we obtain a simplified equation:

$$\Omega_{PABC} = \frac{1}{3} \bar{x}_{(P)} \cdot d\bar{S}_{ABC} \tag{3.10}$$

Volume of a Pyramid

In a similar way, for a Pyramid of vertices $PABCD$ we obtain:

$$\Omega_{PABCD} = \frac{1}{3}\oint_S \bar{x} \cdot d\bar{S} = \frac{1}{3}\bar{x}_{(P)} \cdot d\bar{S}_{ABCD} \tag{3.11}$$

Where the face's normal vector $\bar{S}_{ABCD}$ can be obtained with equation 3.4. Instead since the vertices A, B, C, D are not necessarily coplanar, we compute $\bar{x}_{(P)}$ with a suitable average:

$$\bar{x}_{(P)} = \frac{1}{4}(\bar{x}_{PA} + \bar{x}_{PB} + \bar{x}_{PC} + \bar{x}_{PD}) \tag{3.12}$$

Volume of a Wedge

We compute the volume of a wedge as the sum of the volume of a tetrahedron and the volume of a pyramid:

$$\Omega_{ABCDEF} = \frac{1}{2}(\Omega_{ADEF} + \Omega_{ABCFE}) \tag{3.13}$$

Volume of a Hexahedron

For a hexahedron of vertices A, B, C, D, E, F, G, H we consider two ways of computing its volume. In the first way we compute it as the sum of the volume of five tetrahedra: $DABE$, $DBCG$, $DEGH$, $DBGE$, $FBEG$:

$$\Omega_{HEX1} = \Omega_{DABE} + \Omega_{DBCG} + \Omega_{DEGH} + \Omega_{DBGE} + \Omega_{FBEG} \tag{3.14}$$

In the second way we compute it as the sum of the volume of other different five tetrahedra: $FACB$, $FAEH$, $FCHG$, $FAHC$, $DACH$:

$$\Omega_{HEX2} = \Omega_{FACB} + \Omega_{FAEH} + \Omega_{FCHG} + \Omega_{FAHC} + \Omega_{DACH} \tag{3.15}$$

For a general hexahedron in which points belonging to the same face are not coplanar, it results that $\Omega_{HEX1} \neq \Omega_{HEX2}$. Therefore, in an attempt to get the best approximation, we compute the volume of a hexahedron as the average of the above formulas:

$$\Omega_{HEX} = \frac{1}{2}(\Omega_{HEX_1} + \Omega_{HEX_2}) \tag{3.16}$$

Form must follow function.

Le Corbusier

Chapter 4

Solvers Framework

4.1 Context

Partial Differential Equation A *partial differential equation* (**PDE**) represents a law of physics. For an exhaustive introduction to PDEs, one can read [45], [110], [101], [43], and [68]. A generic PDE can be written as equation 4.1:

$$\mathcal{P}(u,g) = F\left(\mathbf{x}, t, u, \frac{\partial u}{\partial t}, \frac{\partial u}{\partial x_1}, \dots, \frac{\partial u}{\partial x_d}, \dots, \frac{\partial^{p_1+\dots+p_d+p_t} u}{\partial x_1^{p_1} \dots \partial x_d^{p_d} \partial t^{p_t}}, g\right) = 0 \tag{4.1}$$

Where u is the unknown function (the dependent variable), $\mathbf{x}$ denotes the $d+1$ space independent variables: $\mathbf{x} = (x_1, \dots x_d)^T$, t is the temporal independent variable, g is a set of data which the PDE depends on, and $p_i \in \mathbb{N}$. We say that equation 4.1 is of order q if q is the maximum derivation order assumed by p_i.
We can have a first classification of PDEs based on their order. A PDE is said to be *quasi-linear*, if the derivatives of maximum order appear only linearly. A PDE is said to be semi-linear, if it is linear and the coefficient of the derivatives of maximum order depends only on $\mathbf{x}$ and t, but not on u. Finally, if there are no terms which are independent from u, then a PDE is said to be homogeneous.
A second classification is based on their mathematical formulation. Let's consider the case of equations of second order:

$$Lu = A\frac{\partial^2 u}{\partial x_1^2} + B\frac{\partial^2 u}{\partial x_1 \partial x_2} + C\frac{\partial^2 u}{\partial x_2^2} + D\frac{\partial u}{\partial x_1} + E\frac{\partial u}{\partial x_2} + Fu = G \tag{4.2}$$

With $A, B, C, D, E, F \in \mathbb{R}$. Note that in equation 4.2 any x_i can represent the temporal variable t. The second classification is based on the sign of the discriminat:

$$\Delta = B^2 - 4AC \tag{4.3}$$

Specifically, we have three kinds of PDEs:

$$\begin{array}{llll} \text{if} & \Delta < 0 & \text{than the PDE is} & \textit{elliptic} \\ \text{if} & \Delta = 0 & \text{than the PDE is} & \textit{parabolic} \\ \text{if} & \Delta > 0 & \text{than the PDE is} & \textit{hyperbolic} \end{array} \tag{4.4}$$

Each kind of PDE can be solved by a specific numerical method. Let's have a look at some exmples.

Elliptic PDE solved with the Galerkin Method Let' consider a weak formulation of an elliptic problem over a domain $\Omega \subset \mathbb{R}^d$:

$$\text{find} \quad u \in V : a(u, v) = F(v) \quad \forall v \in V \tag{4.5}$$

Where V is an Hilbert Space, and $a(\cdot, \cdot)$ a bilinear form. The Galerkin method consists of finding the approximate solution $u_h \in V_h$, where $V_h \in V$ is a family of spaces which depends on a positive parameter h, and with dimension $dim\, V_h = N_h < \infty$. The discretized problem is then:

$$\text{find} \quad u_h \in V_h : a(u_h, v_h) = F(v_h) \quad \forall v_h \in V_h \tag{4.6}$$

Assuming a base $\phi_j, j = 1, \ldots .N_h$ for V_h, it results that equation 4.7 has to be satisfied for each basis function ϕ_i, because every function of the space V_h can be obtained as a linear combination of the ϕ_is. Because $u_h \in V_h$, then:

$$u_h(\mathbf{x}) = \sum_{j=1}^{N_h} u_j \phi_j(\mathbf{x}) \tag{4.7}$$

Therefore we can recast equation 4.7 as:

$$\sum_{j=1}^{N_h} u_j a(\phi_j, \phi_i) = F(\phi_i), \quad i = 1, \ldots, N_h \tag{4.8}$$

Denoting with A the stiffness matrix with elements $a_{ij} = a(\phi_j, \phi_i)$, and with $\mathbf{f}$ the vector with components $f_i = F(\phi_i)$, the equations 4.8 are equivalent to the following linear system:

$$A\mathbf{u} = \mathbf{f} \tag{4.9}$$

From the above analysis we infer that, from a computational point of view, the ingredients to numerically solve the problem 4.5 with the Galerkin method are:

- A module implementing a Galerkin space discretization method.
- A module implementing a suitable linear system solver.

Parabolic PDE solved with the Galerkin Method Let's consider the following parabolic equation:

$$\frac{\partial u}{\partial t} + Lu = f \quad \mathbf{x} \in \Omega, \quad t > 0 \tag{4.10}$$

Where Ω is the domain, $f = f(\mathbf{x}, t)$ is a given function, $L = a(u, v)$ is a generic elliptic operator that acts on the unknown variable $u = u(\mathbf{x}, t)$. In order to solve equation 4.10, let's first multiply it by a test function $v = v(\mathbf{x})$ and then integrate it over Ω. After that, as we did for problem 4.5, we consider always a Galerkin approximation so that we obtain:

$$\int_\Omega \frac{\partial u_h}{\partial t} d\Omega + a(u_h, v_h) = F(v_h) \quad \forall v_h \in V_h \tag{4.11}$$

Considering:

$$u_h(\mathbf{x}, t) = \sum_{j=1}^{N_h} u_j(t)\phi_j(\mathbf{x}) \tag{4.12}$$

Equation 4.11 can be recast as:

$$\sum_{j=1}^{N_h} \frac{\partial u_j(t)}{\partial t} \underbrace{\int_\Omega \phi_j \phi_i d\Omega}_{m_{ij}} + \sum_{j=1}^{N_h} u_j(t) \underbrace{a(\phi_j, \phi_i)}_{a_{ij}} = \underbrace{F(\phi_i)}_{f_i(t)} \quad i = 1, \ldots, N_h \tag{4.13}$$

Or in matrix form:

$$M\frac{\partial \mathbf{u}(t)}{\partial t} + A\mathbf{u}(t) = \mathbf{f} \tag{4.14}$$

From the above analysis we infer that, from a computational point of view, the ingredients to numerically solve problem 4.10 with Galerkin method are:

- A module implementing a Galerkin space discretization method.
- A module implementing a time discretization method.
- A module implementing a suitable linear system solver, in case we solve equation 4.14 with an implicit time method.

Hyperbolic PDE solved with the Finite Volume Method Let's consider the following hyperbolic PDE:

$$\frac{\partial \mathbf{u}}{\partial t} + \frac{\partial \mathbf{F}(\mathbf{u})}{\partial \mathbf{x}} = \mathbf{S} \tag{4.15}$$

Using the Gauss theorem, equation 4.15 can be integrated over each cell Ω_i that discretizes the domain Ω:

$$\frac{\partial}{\partial t}\int_{\Omega_i}\mathbf{u}\cdot d\mathbf{x}+\oint_{\partial\Omega_i}\mathbf{F}(\mathbf{u})\cdot\mathbf{n}dS=\int_{\Omega_i}\mathbf{Source}\cdot d\mathbf{x}\quad\forall\Omega_i\in\Omega\quad i=1,\ldots,N \tag{4.16}$$

Using the mean value theorem, replacing integrals by sums and analytical fluxes $\mathbf{F}$ with numerical fluxes $\tilde{\mathbf{F}}$ we get the finite volume spatial discretization:

$$\frac{\partial\mathbf{u}_i}{\partial t}+\mathbf{R}_i(\mathbf{u})=0 \tag{4.17}$$

with:

$$\mathbf{R}(\mathbf{u}_i)=\frac{1}{\Omega_i}\sum_{j=1}^{n}\tilde{\mathbf{F}}_j\cdot\mathbf{n}_j dS_j-Source_j \tag{4.18}$$

From the above analysis we infer that, from a computational point of view, the ingredients to numerically solve the problem 4.15 with the finite volume method are:

- A module implementing a Finite Volume space discretization method.
- A module implementing a time discretization method.
- A module implementing a suitable linear system solver, in case we solve equation 4.17 with an implicit time method.

Computational Modules From what has been discussed in the previous paragraphs, it appears clear that the modules for space discretization, time discretization and linear system solving are the essential blocks for a solvers' framework. Each module implements a specific numerical method. Our module implementation policy has been tailored to the following targets.

- Splitting physics from numerics.
- Enabling module interoperability.
- Independent implementation of modules.

In the next paragraphs we will describe how we have addressed their implementation.

Splitting physics from numerics In tab. 4.1 are shown the connections permitted between some numerical methods and some physical models. We see that the same numerical method can be used to discretize different physical models.
As long as mathematically feasible, we want to have multiple discretization methods applicable to any physical model. In this way we can reuse the implementation of a numerical method for different physical models without having to reimplement it. This implies that the numerical methods are not directly coupled with a physical model, but we let them interoperate through

	Navier-Stokes	MHD	Heat Transfer	Elasticity
Finite Volume Method	yes	yes	yes	no
Finite Element Method	yes	yes	yes	yes
Fluctuation Splitting Method	yes	yes	no	no

Table 4.1: Applicability of some Numerical Methods to some Physical Models

an interface.

Independent implementation of modules We have grouped all the modules in three groups:

- *Space Discretization Method Modules.* We called this group **ISM**. It comprises all space discretization methods.
- *Time Discretization Method Modules.* We called this group **IODE**. It includes all time discretization methods.
- *Linear System Solver Modules.* We called this group **INLS**. It gathers all the methods for solving matrix systems.

Each group has its own *abstract interface*, through which its modules can interoperate with the modules of the other groups. Apart from the abstract interface, each module is free from interface contraints in its internal implementation, so that it can be tailored only for its specific purpose. In this way each module can be implemented with its algorithms as the only focus.

Enabling module interoperability Module interoperability is accomplished with a single simple abstract interface for each group of modules. This interface allows us to dynamically connect any module to any other. We have discarded complicated object oriented patterns for module interoperability, like the one of [93], because, in our opinion, they tend to shadow their real purpose, generating confusion. We want form to follow function, not viceversa. Our alternative to OOP is the direct connection of two modules, without further objects which act as a bridge. A module programmer needs only to create a class that derives from the module's group interface.

4.2 Solvers Framework Structure

In fig. 4.1 is depicted the UML diagram of our solvers framework. We now describe how it works. *Abstract Solver* is an object, *ASO*, from which every *Concrete Solvers* has to derive. Currently there are implementations of concrete solvers for Euler Equations and Laminar Navier-Stokes Equations. The ASO manages the data storage and the functions common to

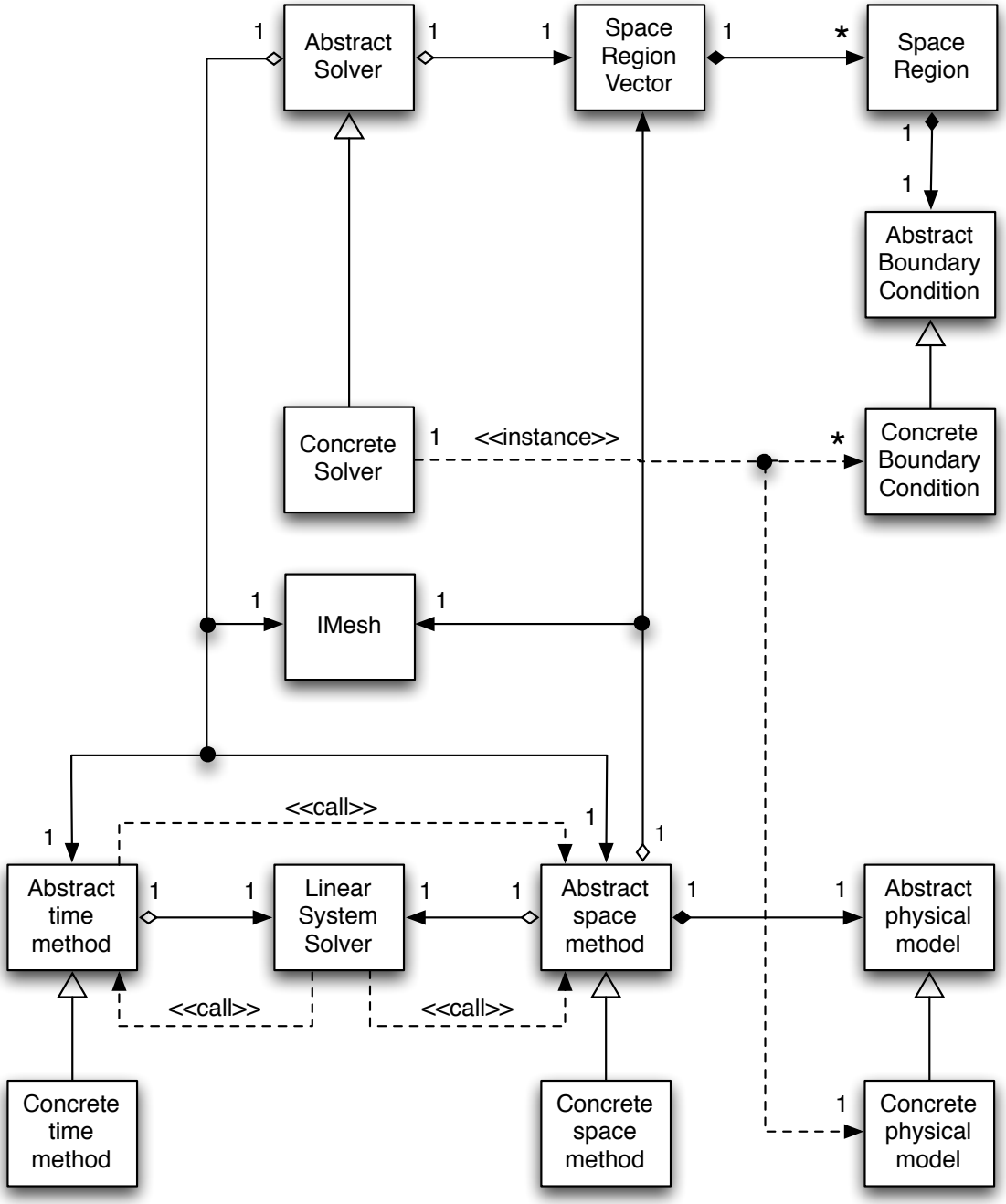


Figure 4.1: UML diagram of our solvers framework

every concrete solver. It has RCPs to the abstract time method object, *ATMO*, and the abstract space method object, *ASMO*. The last two are interfaces to concrete time and space discretization methods (IODE and ISM modules) respectively. Currently the Backward Euler, Forward Euler and Crank-Nicholson time discretization methods are implemented. A Finite Volume Method is, for the moment, the only concrete space discretization method implemented. Both ATMO and ASMO have acquaintance of the interface to INLS, *NLSO*. They can therefore perform direct calls to NLSO functions. On the other hand, NLSO can perform indirect calls (*callbacks*) to both ATMO and ASMO. We will describe this callback mechanism in the next paragraphs. ASMO and ASO share through RCPs an IMesh object and a *space region vector*. With the RCP to IMesh, a concrete space method is able to set up things like its favourite mesh representation, and it has access to the data structure through *iterator objects*. The Space Region vector is an STL vector containing instances of the *SpaceRegion* class. A space region is a portion of the computational domain (as explained in chapter 3), which can represent even a boundary zone. Therefore each space region contains an RCP to an abstract boundary condition, *ABC*. ABC is an interface from which every concrete boundary condition must derive. The task of instantiating a concrete boundary condition is accomplished by the concrete solver. ASMO has also an RCP to an abstract physical model object, APMO. From APMO every concrete physical model must derive. Example of concrete models are: the Euler model and the laminar Navier-Stokes model. The concrete physical models are instantiated by the concrete solver and then shared with the concrete space method. Inside each concrete solver are implemented all the computations needed to perform a simulation; we shall not go into technical details here, but in chapter 6 we will give some further explanations.
ASO, ATMO, ASMO, NLSO, ABC and APMO are all derived from the Configurator class (see paragraph 2.7), so that each one of them can independently and directly set itself up from the XML input parameters file.
All connections between the abstract interfaces are set-up at run time, according to the needs of the user, that passes its requests through the XML input parameters file. Therefore the structure of the code during execution is completelly different from that established at compile time, giving a light a flexible structure.

4.3 Time Method Module

4.3.1 Time Integration Methods

Let's consider a matricial ordinary differential equation (ODE) of the form:

$$\frac{\partial \mathbf{U}}{\partial t} + \mathbf{R}(\mathbf{U}) = 0 \tag{4.19}$$

Where U is the vector of the unknown variables, and $\mathbf{R}$ is the space residual. Equation 4.19 can be solved through an explicit or through an implicit method. Explicit methods do not involve special computational issues, so let's skip them for the moment to concentrate on the implementation of implicit methods.
We define as pseudo-steady residual, $\tilde{R}$:

$$\tilde{\mathbf{R}}(\mathbf{U}) = \frac{\partial \mathbf{U}}{\partial t} + \mathbf{R}(\mathbf{U}) = 0 \tag{4.20}$$

The concrete expression of $\tilde{\mathbf{R}}$ depends on the chosen implicit time integration method. We have considered the two following ones:

$$\text{Backward Euler (BE)} \qquad \tilde{\mathbf{R}}(\mathbf{U}) = \frac{\mathbf{U} - \mathbf{U^n}}{\Delta t} + \mathbf{R}(\mathbf{U}) = 0 \tag{4.21}$$

$$\text{Crank Nicolson (CN)} \qquad \tilde{\mathbf{R}}(\mathbf{U}) = \frac{\mathbf{U} - \mathbf{U^n}}{\Delta t} + \frac{1}{2}(\mathbf{R}(\mathbf{U}) + \mathbf{R}(\mathbf{U^n})) \tag{4.22}$$

Expanding the pseudo-steady residual, $\tilde{\mathbf{R}}(\mathbf{U})$, to the Taylor series around the current time solution $\mathbf{U^n}$, and considering that $\tilde{\mathbf{R}}(\mathbf{U^{n+1}}) = 0$ must be satisfied, we obtain:

$$\tilde{\mathbf{R}}(\mathbf{U^{n+1}}) = \tilde{\mathbf{R}}(\mathbf{U^n}) + \left[\frac{\partial \tilde{\mathbf{R}}}{\partial \mathbf{U}}(\mathbf{U^n})\right]\Delta \mathbf{U^n} = 0 \tag{4.23}$$

Applying the Newton method to equation 4.23, we get an iterative procedure in which we have to solve a linear system inside each iteration:

$$\begin{cases} \left[\frac{\partial \tilde{\mathbf{R}}}{\partial \mathbf{U}}(\mathbf{U^k})\right]\Delta \mathbf{U^k} &= -\tilde{\mathbf{R}}(\mathbf{U^k}) \\ \tilde{\mathbf{R}}(\mathbf{U^{k+1}}) &= \tilde{\mathbf{R}}(\mathbf{U^k}) + \Delta \mathbf{U^k} \end{cases} \tag{4.24}$$

Where $\mathbf{U^{k=0}} = \mathbf{U^n}$. In a steady computation, the Newton procedure is directly stopped after one Newton sub-iteration, so that: $\mathbf{U^{k=1}} = \mathbf{U^{n+1}}$. In an unsteady computation, the Newton sub-iterations stop when $\|\Delta \mathbf{U^k}\| < \epsilon$, and then we set: $\mathbf{U^{n+1}} = \mathbf{U^{k^{last}+1}}$.
The term $\left[\frac{\partial \tilde{\mathbf{R}}}{\partial \mathbf{U}}(\mathbf{U^k})\right]$ is the Jacobian matrix, J:

$$J = \left[\frac{\partial \tilde{\mathbf{R}}}{\partial \mathbf{U}}(\mathbf{U^k})\right] = \left[\underbrace{\frac{\partial(\partial \mathbf{U}/\partial t)}{\partial \mathbf{U}}(\mathbf{U^k})}_{J_t} + \underbrace{\frac{\partial \mathbf{R}}{\partial \mathbf{U}}(\mathbf{U^k})}_{J_R}\right] = J_t + J_R \tag{4.25}$$

Where J_t is the temporal jacobian and it is related to the time discretization, conversely J_R is the residual jacobian and it is related to the spatial discretization. Usually J_t can be easily calculated, for example for both Backward Euler and Crank-Nicholson time discretization, we have:

$$J_t = \frac{I}{\Delta t} \tag{4.26}$$

A major problem arises instead for J_R. Normally it is computed either analytically or numerically. The former implies that we should derive all expressions that arise from the spatial

discretization; nevertheless these expressions can be quite complicated, and so it results that the analytical method is ultimately tiresome, cumbersome and error-prone. The easiest way is the numerical method, where J_R is computed via finite differences. This can be a good compromise solution, but sometimes it can be useful to have a more precise computation of the derivatives in order to exploit the quadratical convergence velocity of the Newton procedure. Having discarded the analytical solution, we have decided on Automatic Differentiation instead. In conclusion, inside this module we can compute J_R either numerically or via Automatic Differentiation.
In case we rely on explicit methods there is obviously no need to use a Newton method and to solve a linear system: the things are quite simpler. But it has the drawback of conditional stability, that is, a restriction on the time step, Δt. Moreover an explicit method is not suitable for stiff problems. Nevertheless, there are situations where it is more advantageous to use an explicit method, and others where it is preferable to use an implicit method. Just to be conservative, we therefore decided to implement the Forward Euler method as well.

4.3.2 Automatic Differentiation

Inside Hydra we have Automatic Differentiation capabilities through either our *AutoDiff* class or the external Sacado library, [11]. In both cases what should be done is to templatize the type of the variables for which the derivative has to be computed. AutoDiff computes derivatives in forward mode using a C++ operator overloading implementation of dual numbers. Here we will give just a brief description on how AutoDiff has been implemented.
Equation 4.27 is the definition of a derivative:

$$f'(x) = \lim_{\delta \to 0} \frac{f(x+\delta) - f(x)}{\delta} \tag{4.27}$$

Let's consider for example the function: $f(x) = 2x^2 + 1$, and an approximation of its derivative with a finite increment d:

$$f'(x) \simeq \frac{f(x+d) - f(x)}{d} = \frac{4dx + 2d^2}{d} = 4x + 2d \tag{4.28}$$

The exact derivative is $4x$ so we have an error term, $2d$, which comes from $2d^2$. We can have a zero error if d were a special number, let's call it *exotic number*, with the exotic property that: $d^2 = 0$. Apart from this exotic property, we suppose d to be endowed with other properties, such as commutativity: $ad = da$ and $a + d = d + a$, where a is a real number (a *double* or a *float* in C++). With the dual number we can rewrite equation 4.27 as:

$$f(x+d) = f(x) + df'(x) \tag{4.29}$$

Let's consider another example: the derivative of $g(x) = x^n + x$ is:

$$g'(x) = nx^{n-1} + 1 \tag{4.30}$$

Now let's define the *dual number* as a real number x extended with an exotic number d:

$$\underbrace{y}_{\text{dual number}} = \underbrace{x}_{\text{real number}} + \underbrace{d}_{\text{exotic number}} \tag{4.31}$$

If we compute $g(y)$ we get:

$$g(y) \equiv g(x+d) = x^n + nx^{n-1}d + d^2\left(\frac{n(n-1)}{2}x^{n-2} + \cdots\right) + d = \underbrace{x^n}_{g(x)} + \underbrace{(nx^{n+1}+1)}_{g'(x)}d \tag{4.32}$$

From equation 4.32 we see that the coefficient of d: $nx^{n+1}+1$, is the exact derivative of $g(x)$, see equation 4.30. We have just defined a strategy for automatic differentiation: perform all computations on the real variable x extended with the exotic variable d; the derivatives will be accumulated as the coefficient of d.
We shall now describe how we implemented the dual number, using two unique features of C++: operator overloading and generic programming (templates). First of all, in Code Listing 4.1, we have defined the class representing the dual number type. In Code Listing 4.1 it is shown just a simplified version.

Listing 4.1: AutoDiff class definition

```
class AutoDiff { public: // Dual number
  double a; // Real number
  double d; // Exotic number
  // Constructor
  AutoDiff(double a0, double d0 = 0.0d) : a(a0), d(d0) { }
};
```

We have overloaded several operators and operations. Here we just show the code listings of some of them. In Code Listing 4.2 is shown the overloading of the addition operator, in Code Listing 4.3 is shown the overloading of the multiplication operator, and in Code Listing 4.4 is shown the overloading of operator cosine.

Listing 4.2: Overloading of operator +

```
AutoDiff operator+(const AutoDiff &x, const AutoDiff &y) {
  return AutoDiff(x.a+y.a, x.d+y.d);
}
```

Listing 4.3: Overloading of operator *

```
AutoDiff operator*(const AutoDiff &x, const AutoDiff &y) {
  return AutoDiff(x.a*y.a, x.a*y.d+x.d*y.a);
}
```

Listing 4.4: Overloading of the cosine trascendental function

```
AutoDiff cos(const AutoDiff &x) {
  return AutoDiff(cos(x.a), -d*sin(x.a));
}
```

We can extract the value of the variable querying for member a, while we can extract the derivative querying for d. In case we want to deal with partial derivatives, the mechanism remains essentially the same, with the only difference that in our implementation we have substituted d with an STL vector, which has a number of elements equal to the number of partial derivatives. With the STL vector we acquire large flexibility because we can set the number of partial derivatives at run time.
In order to use AutoDiff, a function must be templatized. For example, the implementation of function $f(x) = (x + 2)(y + 1)$ would be as shown in Code Listing 4.5. We then can use it as in Code Listing 4.6.

Listing 4.5: Implementation of a simple function

```
template<typename ScalarT> ScalarT func(ScalarT x, ScalarT y) {
  return (x + ScalarT(2.0d))*(y + ScalarT(1.0d));
}
```

Listing 4.6: Use of AutoDiff for a simple function

```
double a = 10.0;
double b = 15.0;
int num_deriv = 2;  // Number of independent variables
AutoDiff x(num_deriv, 0, a); // First (0) independent variable
AutoDiff y(num_deriv, 1, b); // Second (1) independent variable
// Compute function and derivative with AutoDiff
AutoDiff z = func(AD1, AD2);
double z_value = z.val();  // Extract value
double dzdx = z.dx(0); // Extract partial derivative dz/da
double dzdy = z.dx(1); // Extract partial derivative dz/db
```

4.3.3 Abstract Interface

In fig, 4.2 is shown the collaboration mechanism we implemented between modules from IODE, ISM and INLS, for the cases of explicit and implicit time integration. For an explicit method the iterative procedure we implemented works as follows:

1. IODE asks ISM to compute the residuals vector, $\mathbf{R}(\mathbf{U^n})$, corresponding to the variables vector, $\mathbf{U^n}$ at time step n.
2. With $\mathbf{R}(\mathbf{U^n})$ IODE computes the variables vector $\mathbf{U}$ either at the next time step $n + 1$, or at the next stage in an explicit multi-stage method. In the former case there will be an interior loop for each stage, in which IODE asks ISM to compute the residual vector corresponding to the variables vector at stage k.

On the other hand, when considering an implicit method the iterative procedure we implemented works as follows:

1. IODE asks ISM to compute the residuals vector, $\mathbf{R}(\mathbf{U^n})$, corresponding to the variables vector, $\mathbf{U^n}$ at time step n.
2. IODE computes $\tilde{\mathbf{R}}(\mathbf{U^n})$ and passes it, together with $\mathbf{U^n}$, to INLS, which in turn will compute $\mathbf{U^{n+1}}$. INLS accomplishes this beginning a Newton iterative procedure:

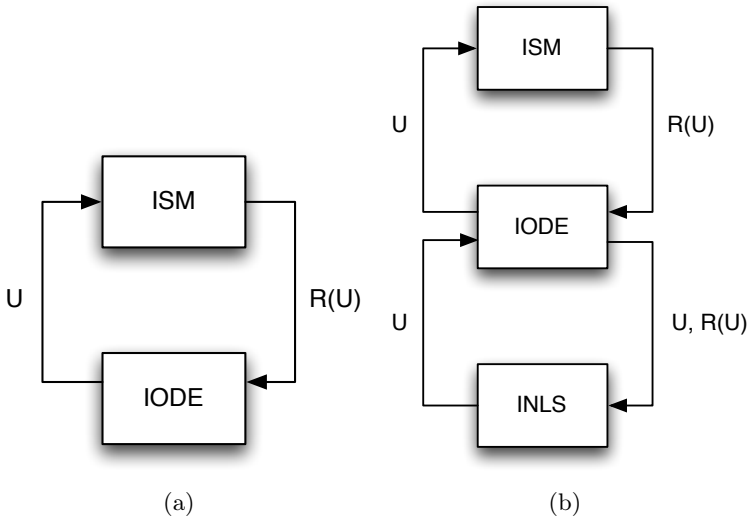


Figure 4.2: Collaboration between modules from IODE, INLS and ISM, for the cases of an explicit (a) or implicit (b) time integration.

(a) INLS computes $\mathbf{U^k}$ and passes it to IODE.

(b) IODE receives $\mathbf{U^k}$ and passes it to ISM.

(c) ISM receives $\mathbf{U^k}$, computes $\mathbf{R}(\mathbf{U^k})$, and passes this result to IODE.

(d) IODE receives $\mathbf{R}(\mathbf{U^k})$, computes $\tilde{\mathbf{R}}(\mathbf{U^k})$, and passes this result to INLS, which in turn will begin a new Newton sub-iteration.

3. Once the Newton procedure has finished, IODE will receive $\mathbf{U^{n+1}}$ from INLS.

The residuals and variables vector are the field vectors described in paragraph 3.3. In a multi-stage method, IODE could need to store more than one vector of variables and/or residual contemporarily. To do this, we gave to IODE the capability to build and store field vector objects on its own.
To carry out the procedure heretofore described, IODE, INLS and ISM should all call functions of each other. Since, as we said, we want that each module can be plugged to any other we cannot statically bind the connection between the modules. Therefore we designed a dynamic binding between modules that makes use of RCP, boost::function and boost::any. RCP is used to let all modules have acquaintance of each other allowing then direct calls between them. Instead boost::function and boost::any are used for indirect calls, or callback functions between them. These callbacks are those calls that take place in the inner loops of a multi-stage method or of the Newton procedure. In Code Listing 4.7 are defined the IODE protected callback functions which bind some ISM functions for the computation of the residuals, the residual jacobian, and of both residuals and residual jacobian. These functions will be set-up by the template functions defined in Code Listing 4.8.
IODE makes available some functions that can be bound by some callbacks in INLS, so that INLS can call IODE to compute pseudo-steady residual vectors corresponding to a sub-iteration of the Newton procedure. In Code Listing 4.9 there are the definitions of these functions. Each

Listing 4.7: IODE functions that bind some ISM functions

```
// Compute space residual
boost::function<void (EVector & U, EVector & RS)> _SpaceResidual;
// Compute jacobian of space residual
boost::function<void (double alpha, double beta,
  EVector & U, EMatrix & JS)> _SpaceJacobian;
// Compute space residual and space jacobian
boost::function<void (double alpha, double beta, EVector & U,
  EVector & RS, EMatrix & JS)> _SpaceResidualJacobian;
// Compute the time step
boost::function<void (double & CFL,
  double & TimeStep)> _SetTimeStep;
```

Listing 4.8: Set-up of the IODE functions that bind some ISM functions

```
template <typename T> void setSpaceResidual (T SR)
   {_SpaceResidual = SR;}
template <typename T> void setJacobian (T J)
  {_SpaceJacobian = J;}
template <typename T> void setSpaceResidualAndJacobian (T SRJ)
  {_SpaceResidualJacobian = SRJ;}
template <typename T> void setTimeStep (T TS) {_SetTimeStep;}
```

concrete time discretization method is encapsulated inside its own class. To make all the above mechanism work, we let the concrete methods' classes derive from the GroupInterface class. Here we will not go into the explanation of all other implementation details to avoid complicated technicalities.

4.4 Space Method Module

The GroupInterface of ISM contains in its protected methods an RCP to:

- An IMesh object, 3.3, so it can have access to all mesh and field data.
- The Abstract Physical Model object, that is explained in paragraph 4.6, so it can have access to the physical model.
- The space region vector, that is explained in paragraph 4.7, so it can have access to the boundary conditions.

In Code Listing 4.10 are shown the definitions of functions that can be bound by either:

- The IODE callback functions, for time-dependent computations.
- The INLS callback functions, for solving, for example, some elliptic problems.

Every concrete space discretization must have a main class that coordinates all the computation and that derives from the ISM Abstract Interface. Each derived concrete method must be implemented in a template fashion with respect to the dependent variable, so that we are able to use automatic differentiation to compute the residual jacobian. In chapter 6 the implementation

Listing 4.9: Definition of IODE functions that can be bound by INLS callbacks functions

```
// Compute Function: F(U)
void PseudoSpaceResidual(EVector & U, EVector & F);
// Compute Jacobian: J=dF(U)/dU
void PseudoSpaceJacobian(EVector & U, EMatrix & J);
// Just get the already computed jacobian
EMatrix getJac();
```

Listing 4.10: Definition of IODE functions that can be bound by INLS callback functions

```
// Compute space residual
void ComputeResidual(EVector & U, EVector & RS);
// Compute jacobian of space
// residual as: JS = alpha + beta*(dRs/dU);
void ComputeJacobian(double alpha, double beta,
  EVector & U, EMatrix & JS);
// Compute space residual and space
// jacobian as: JS = alpha + beta*(dRs/dU)
void ComputeResidualJacobian(double alpha, double beta,
  EVector & U, EVector & RS, EMatrix & JS);
// Compute the time step
void ComputeTimeStep(double & CFL, double & TimeStep);
```

of a concrete space discretization method is described at length: the finite volume method. So we refer to that chapter for further details on ISM.

4.5 Linear System Module

INLS includes a set of classes to interface Hydra with the external linear system solving libraries PETSc and Trilinos. It is through this interface that we can solve linear and non linear system with modern techniques, like Krilov methods. Exhaustive books on the solution of linear systems are [99],[98],[50] and [40].
In C.L. 4.11 are shown the callback functions that can bind either IODE functions or ISM functions.

Listing 4.11: Definition of INLS callback functions

```
// Compute Function: F(U)
boost::function<void ()> _FF;
boost::function<void (EVector & U, EVector & F)> _Function;
// Compute Jacobian: J=dF(U)/dU
boost::function<void (EVector & U, EMatrix & J)> _Jacobian;
// Just get the already computed jacobian
boost::function<EMatrix ()> _Jac;
```

These functions will be set-up by the template functions defined in C.L. 4.12. We shall skip here all internal implementation details of the interface classes, as it is quite technical.

Listing 4.12: Set-up of INLS callback functions

```
// Compute F(U)
template <typename T> void setFunction (T F) {_Function = F;}
// Compute jacobian dF(U)/dU
template <typename T> void setJacobian (T J) {_Jacobian = J;}
// just get the already computed jacobian
template <typename T> void setJac (T J) {_Jac = J;}
```

4.6 Physical Models

The physical model abstract interface provides the definitions of all the functions needed to describe the physics of the simulated phenomena. The actual implementation of these functions is accomplished inside the derived classes, each one corresponding to a particular physics. Every function must be templatized with respect to the variables' type, in order to apply automatic differentiations. Basically the template parameters can be instantiated either as *double* or AutoDiff.

4.7 Boundary Conditions

Our approach to implement the boundary conditions consists of using ghost entities. These entities can either be cells or vertices, as the field variables can be associated with anyone of the two. In our implementation, when a concrete space method reaches the domain's boundary, during its iterations over the cells, or vertices, it will ask for a vector of variables which represent the outside field conditions. We call this vector *boundary state vector* (BSV); it stores the same physical variables stored in the field containers, so that the concrete space method can treat the boundaries as if they were internal to the domain. The BSV is built on flight, that is, at execution time when needed, by a concrete boundary conditions object (CBCO). Basically when the concrete space method reaches an entity that lies on the domain's boundary (boundary entity) it queries for:

- the boundary entity space region;
- the coordinate of the boundary entity;
- the coordinate of some entities close to the boundary entities (for high order boundary reconstruction);
- the field vector of the boundary entity.

With such information, a CBCO can compute the BSV corresponding to the boundary condition variable values (BCVV). The BCVV often represent physical variables different from those of the field vectors, therefore the CBCO has to convert them to the field vector variables. The BCVV are represented as third order polynomial in cartesian space coordinate independent variables. In this way we are able to set-up non uniform boundary conditions. Basically the user has the possibility to set-up the coefficients of the polynomials up to the third order, if he/she does not set a coefficient, then it will be automatically set-up to the default zero value.

Part II

Fluid Flow Modeling and Simulation

Preamble of Part II

Computational Fluid Dynamics (CFD) is the discipline which deals with the simulation of fluid flow phenomena. Its origins date back to the 1917, when the British scientist L. F.Richardson attempted for the first time to solve numerically the Navier-Stokes equations in order to predict the weather. His attempt was made by hand: with pencil and paper! Nowadays pencil and paper have been substituted by more and more sophisticated computers , and CFD has grown with the development of elaborate numerical methods and algorithms.
The role of CFD in research and development is continuously increasing, leading to encounter new problems and seek ways to resolve them. Therefore algorithms, numerical methods and programming strategies are in constant evolution. Sometimes it becomes difficult to have an updated overview of the current status of the research and of the many different tools we have at our hand. Things can become dramatically difficult when we have to select the most recent and favorable method for a given application.
In this second part of the thesis the development of our CFD solver is described. This solver is able to solve both compressible Euler equations and compressible Laminar Navier-Stokes equations (it implements density-based methods) on 3D unstructured meshes with parallel computation. First of all we will discuss the mathematical description of fluid flows, then we will adress the numerical modellization in a Finite Volume framework, we will show how we implemented the solver, and finally we will report some simulations as validation testcases.

Do not multiply objects without necessity.

W. Occam

Chapter 5

Fluid Flow Modeling

5.1 The Fluid Flow and its Mathematical Description

The partial differential equation system that describes the motion of fluid flow is called *Navie-Stokes Equations* (**NSE**). NSE are a system of non-linear conservation laws which reflect the fundamental physical principles of conservation of mass, energy and momentum. For an exaustive introduction to the NSE we suggest the following books: [19], [18], [17], [56], [87], [20] and [27].
NSE are often written in the so called *divergence* or *conservative* form, which in a *differential formulation* it is expressed as:

$$\frac{\partial \mathbf{U}}{\partial t} + \nabla \cdot \mathbf{F}_c(\mathbf{U}) - \nabla \cdot \mathbf{F}_v(\mathbf{U}, \nabla \mathbf{U}) = Q_v(\mathbf{U}) + \nabla \cdot \mathbf{Q}_s(\mathbf{U}) \tag{5.1}$$

Whereas in the *integral formulation*, it is expressed as:

$$\frac{\partial}{\partial t} \int_\Omega \mathbf{U} d\Omega + \oint_{\partial\Omega} \Big(\mathbf{F}_c(\mathbf{U}) - \mathbf{F}_v(\mathbf{U}, \nabla \mathbf{U}) \Big) d\mathbf{S} = \int_\Omega Q_v(\mathbf{U}) d\Omega + \oint_{\partial\Omega} \mathbf{Q}_s \cdot d\mathbf{S} \tag{5.2}$$

Where $\mathbf{U}$ denotes the vector of conservative variables, $\mathbf{F}_c(\mathbf{U})$ is the convective tensor flux function, $\mathbf{F}_v(\mathbf{U}, \nabla \mathbf{U})$ is the viscous tensor flux function, Q_v is the volume source term, and finally $\mathbf{Q}_s(\mathbf{U})$ is the surface source term.
For *Computational Fluid Dynamics* purposes (CFD), the integral formulation of the NSE, equation 6.1, has two very important and desirable properties, that makes it advantageous respect to the differential form:

1. If there are no volume sources present, the variation of $\mathbf{U}$ depends solely on the flux across the boundary $\partial\Omega$ and not on any flux inside the control volume Ω.

2. This particular form remains valid in the presence of discontinuities in the flow field like shocks or contact discontinuities.

Let's now explain the meanings of the terms appearing in equations 5.1 and 6.1. The term **U** denotes the vector of conservative variables, and in three dimensional Cartesian reference system it reads as:

$$\mathbf{U} = \begin{pmatrix} \rho \\ \rho u \\ \rho v \\ \rho w \\ \rho e_0 \end{pmatrix} \tag{5.3}$$

Where u, v, and w denotes the Cartesian components of the velocity vector $\mathbf{v}$. ρ is the density of the fluid. e_0 is the total (or stagnation) energy per unit mass: $e_0 = e + \|\mathbf{v}\|_2/2$.
The convective tensor flux is defined as:

$$\mathbf{F}_c = \begin{pmatrix} \rho u & \rho v & \rho w \\ \rho u^2 + p & \rho uv & \rho uw \\ \rho vu & \rho v^2 + p & \rho vw \\ \rho wu & \rho wv & \rho w^2 + p \\ \rho u h_0 & \rho v h_0 & \rho w h_0 \end{pmatrix} \tag{5.4}$$

While the viscous tensor flux is defined as:

$$\mathbf{F}_v = \begin{pmatrix} 0 & 0 & 0 \\ \tau_{xx} & \tau_{xy} & \tau_{xz} \\ \tau_{yx} & \tau_{yy} & \tau_{yz} \\ \tau_{zx} & \tau_{zy} & \tau_{zz} \\ f_{ex} & f_{ey} & f_{ez} \end{pmatrix} \tag{5.5}$$

With the energy flux component f_{ex}, f_{ey}, and f_{ez} defined as:

$$\begin{cases} f_{ex} = u \cdot \tau_{xx} + v \cdot \tau_{yx} + w \cdot \tau_{zx} - q_x \\ f_{ey} = u \cdot \tau_{xy} + v \cdot \tau_{yy} + w \cdot \tau_{zy} - q_y \\ f_{ez} = u \cdot \tau_{xz} + v \cdot \tau_{yz} + w \cdot \tau_{zz} - q_z \end{cases} \tag{5.6}$$

The heat flux vector components q_k are given by:

$$\begin{cases} q_x = -k \frac{\partial T}{\partial x} \\ q_y = -k \frac{\partial T}{\partial y} \\ q_z = -k \frac{\partial T}{\partial z} \end{cases} \tag{5.7}$$

The shear stress tensor components τ_{ij} are given by:

$$\begin{cases} \tau_{xx} = 2\mu\frac{\partial u}{\partial x} + \lambda\left(\frac{\partial u}{\partial x} + \frac{\partial v}{\partial y} + \frac{\partial w}{\partial z}\right) \\ \tau_{yy} = 2\mu\frac{\partial v}{\partial y} + \lambda\left(\frac{\partial u}{\partial x} + \frac{\partial v}{\partial y} + \frac{\partial w}{\partial z}\right) \\ \tau_{zz} = 2\mu\frac{\partial w}{\partial z} + \lambda\left(\frac{\partial u}{\partial x} + \frac{\partial v}{\partial y} + \frac{\partial w}{\partial z}\right) \end{cases} \tag{5.8}$$

and

$$\begin{cases} \tau_{xy} = \tau_{yx} = \mu\left(\frac{\partial u}{\partial y} + \frac{\partial v}{\partial x}\right) \\ \tau_{yz} = \tau_{zy} = \mu\left(\frac{\partial v}{\partial z} + \frac{\partial w}{\partial y}\right) \\ \tau_{zx} = \tau_{xz} = \mu\left(\frac{\partial w}{\partial x} + \frac{\partial u}{\partial z}\right) \end{cases} \tag{5.9}$$

As one can see the NSE are a system of five PDE, whose equation when written as above have the meaning of:

1. Equation 1: the *continuity equation*, it expresses the conservation of the mass.
2. Equation from 2 to 4: the *momentum equations*, they express the conservation of the momentum in Cartesian directions x, y, and z, respectively.
3. Equation 5: the *energy equation*, it expresses the conservation of the energy.

When the influence of viscous shear stresses and heat conduction effects can be neglected, the NSE assume a form called the *Euler Equations*:

$$\frac{\partial \mathbf{U}}{\partial t} + \nabla \cdot \mathbf{F}_c(\mathbf{U}) = Q_v(\mathbf{U}) + \nabla \cdot \mathbf{Q}_s(\mathbf{U}) \tag{5.10}$$

In equation 5.10 the terms $\mathbf{U}$ and $\mathbf{F}_c(\mathbf{U})$ are those defined by equations 5.3 and 5.5 respectivelly. In order to close the system of PDE we have to augment the NSE with algebraic expressions which relates the internal energy, e, the pressure, p, the dynamic viscosity, λ, and the thermal conductivity coefficient, k, to the thermodynamic state of the fluid:

$$p = p(T, \rho) \qquad e = e(T, \rho) \qquad \mu = \mu(T, \rho) \qquad \lambda = \lambda(T, \rho) \qquad k = k(T, \rho) \tag{5.11}$$

Equations 5.11 might have a very complex actual expression. In same cases can be assumed the hypothesis of perfect gases. For a polytropic ideal gas the pressure can be computed as:

$$p(\rho, \rho e, \rho \mathbf{v}) = (\gamma - 1)\left(\rho e_0 - \frac{1}{2}\frac{\rho \mathbf{v} \cdot \rho \mathbf{v}}{\rho}\right) \tag{5.12}$$

where $\gamma = C_p/C_v$ is the substance dependent ratio of specific heat capacities. The dynamic viscosity coefficient μ is a function of both temperature and density, however for an ideal

gas it is usually approximated only in function of the temperature. Example of this type of approximation are the Sutherland formula:

$$\frac{\mu}{\mu_0} = \frac{T_0 + C}{T + C}\left(\frac{T}{T_0}\right)^{\frac{3}{2}} \tag{5.13}$$

or the power law equation:

$$\frac{\mu}{\mu_0} = \left(\frac{T}{T_0}\right)^{\alpha} \tag{5.14}$$

Where the coefficient T_0, μ_0 and α depend on the substance. The second viscosity coefficient λ is often computed in function of the first viscosity parameter μ, by the Stokes hypothesis:

$$\lambda(T) = -\frac{2}{3}\mu(T) \tag{5.15}$$

The thermal conductivity coefficient k can be computed as a function of μ assuming a constant Prandtl number:

$$k(T) = \frac{C_p}{Pr}\mu(T) \tag{5.16}$$

The solution of the Navier-Stokes equations 5.1 or 6.1 does not raise any fundamental difficulties in the case of laminar flow. The simulation of turbulent flows, however, presents substantial problems. An exhaustive treatment of the argument is made [90] and [120].

5.2 The Boundary Conditions

We now briefly review the application of the boundary condition for the Euler and Navier-Stokes Equations. For an exaustive introduction to this topic we suggest the books: [50], [62], [107], [124], [103] and the articles: [86], [35] and [47].
The Euler equations are a time-dependent hyperbolic system with four unknown dependent variables. We have to know how to handle these four variables at the domain boundary in order to define suitable boundary conditions. Basically, the boundaries can be grouped in three types: *inlet*, *outlet* and *wall*; each one requiring a proper treatment.
First of all let's consider the system of Euler equations in 2 dimensions. Due to the hyperbolic nature of this system, we can study the treatment of the boundary conditions with a characteristic analysis. We know that this system have four eigenvalues which correspond to the following four propagation speeds:

$$\begin{cases} \lambda_1 = \frac{\mathbf{v}\cdot\mathbf{k}}{\|\mathbf{k}\|_2} \\ \lambda_2 = \frac{\mathbf{v}\cdot\mathbf{k}}{\|\mathbf{k}\|_2} \\ \lambda_3 = \frac{\mathbf{v}\cdot\mathbf{k}}{\|\mathbf{k}\|_2} + c \\ \lambda_4 = \frac{\mathbf{v}\cdot\mathbf{k}}{\|\mathbf{k}\|_2} - c \end{cases} \tag{5.17}$$

Where $\mathbf{k}$ is the wave number vector. In a multidimensional domain the transport properties at a surface are determined by the normal component of the fluxes, that is, the Jacobian matrix

associated with the normal vector to the boundary defines the number and type of boundary conditions, we thus define the: **quasi-one dimensional propagation properties** as:

$$\begin{cases} \lambda_1 = \mathbf{v} \cdot \mathbf{n} = \mathbf{v_n} \\ \lambda_2 = \mathbf{v} \cdot \mathbf{n} = \mathbf{v_n} \\ \lambda_3 = \mathbf{v} \cdot \mathbf{n} + c = \mathbf{v_n} + c \\ \lambda_4 = \mathbf{v} \cdot \mathbf{n} - c = \mathbf{v_n} - c \end{cases} \tag{5.18}$$

Where $\mathbf{n}$ is the inward boundary normal vector. In equation 5.18, the first eigenvalue is associated to the vorticity, while the second is associated to the entropy. The last two eigenvalues are associated to the acoustic waves. These eigenvalues define the speed with which the flow information conveys in the (n, t) plane, that is:

$$\frac{dn}{dt} = \lambda \tag{5.19}$$

Thus the sign of λ determines if the information is propagated inward or outward in the computational domain. Therefore we can state that:

- When the sign of λ is positive then an information is propagated from outside toward the inside of the computational domain; it means that this information has to be definded from outside.
- When the sign of λ is negative then an information is propagated from inside toward the outside of the computational domain; it means that this information has to be definded from inside.

Let's analyse some concrete cases.

Inlet For a subsonic inlet three eigenvalues are positive and one is negative, therefore three quantities have to be imposed on the boundary, while a fourth one has to be determined from the interior conditions. Conversely for a supersonic inlet all the eigenvalues are positive and thus all four quantities have to be imposed on the boundary.

Outlet For a subsonic outlet three eigenvalues are negative and one is positive, therefore one quantity has to be imposed on the boundary, while the other three have to be determined from the interior conditions. Conversely for a supersonic outlet all the eigenvalues are negative and thus no quantities have to be imposed on the boundary, but they all have to be determined from the interior.

Wall At a solid wall boundary the normal velocity is zero because no flux can penetrate the solid body. Hence, only one eigenvalue is positive, so that we must impose only one quantity on the bounday.

If we now consider the presence of viscous and thermal fluxes we recover the time-dependent Navier-Stokes equations, which are of a mixed parabolic-hyperbolic nature. This nature has an

impact on the number and type of boundary conditions. For high Reynolds number flow the inlet and outlet boundary conditions can be applied as for the Euler equations. For not high Reynolds number flow things became more complicated, but without going into the details (we refer the interested reader to [49], [35] [47]), if the inlet and outlet boundaries are far enough from the walls, we fall back on the same use as for the Euler equations, while for wall boundary condition of Navier-Stokes equations we have to impose conditions on:

- The velocity: the relative velocity between flow and wall must vanish, this is the no-slip boundary condition.
- The temperature: we can mainly have:
  - Adiabatic wall, there is no heat flux through the wall, this is expressed by the von Neumann condition: $\frac{\partial T}{\partial n} = 0$.
  - Constant temperature wall, we impose the Dirichlet condition: $T = T_w$.
  - Imposed heat flux: $\frac{\partial T}{\partial n} = q_e$.

5.3 Numerical Methods for Fluid Flow Equations

The most adopted approach to solve numericaly the Euler or Navier-Stokes equations is that of discretizing separately the space and the time. It is a two step approach, also called the *method of line*, which consists in:

1. Discretize in space the system of partial differential equations in order to obtain a system of ordinary differential equations.
2. Discretize in time the above obtained system of ordinary differential equations.

The main numerical methods which nowadays are at our disposal to discretize in space the Euler or Navier-Stokes equations are the followings:

- *Finite Difference Method.* Many books have been published on this subject, we personally suggest [63].
- *Finite Volume Method.* As reference we suggest [62], [41], [49], [107], [76] and [50]
- *Finite Element Method.* As reference we suggest [124], [40], [34], [103] and [51], [28].
- *Discontinuous Galerkin Method.* This is a recent developed strategy, just few books have so far been published, here we cite [64], [95].
- *Residual Distribution Sheme.* This is also a recent developed strategy, we consider that [94] is a good reference.
- *Spectral Method.* An introduction can be found in [89].
- *Meshfree Method.* This is a method whose skills are still uncertain. Its main reference are [74], and [97].

For the time discretization instead, a long list of strategies have been developed, mainly divided in implicit or explicit method.

Concepts that have proven useful in ordering things easily achieve such authority over us that we forget their earthly origins and accept them as unalterable givens.

A. Einstein

Chapter 6

Fluid Flow Solver Development

6.1 Context

We implement a solver for Euler and Navier-Stokes equations, using the method of line, where we use Finite Volume for the space discretization, and various explicit or implicit methods for time discetization. The time discretization argument has already been cooped with in paragraph 4.3, therefore we will only coope with the space discretization strategy in the following.
As space discretization strategy we rely on Finite Volume method. Finite Volume Methods are based on the integral form of the Navier-Stokes equations:

$$\frac{\partial}{\partial t}\int_{\Omega}\mathbf{U}d\Omega+\oint_{\partial\Omega}\Big(\mathbf{F}_c(\mathbf{U})-\mathbf{F}_v(\mathbf{U},\nabla\mathbf{U})\Big)d\mathbf{S}=\int_{\Omega}Q_v(\mathbf{U})d\Omega+\oint_{\partial\Omega}\mathbf{Q}_s\cdot d\mathbf{S} \tag{6.1}$$

At the beginning of the method we have to break the domain Ω into the set Γ of finite number N of subdomains or mesh cells or volumes, Ω_i. These volumes have to satisfy the following conditions:

$$\Omega_i\cup\Omega_j=0 \qquad \forall\,\Omega_i,\Omega_j\in\Gamma,\quad i\neq j \tag{6.2}$$

Therefore we have:

$$\frac{\partial}{\partial t}\int_{\Omega_i}\mathbf{U}d\Omega+\oint_{\partial\Omega_i}\Big(\mathbf{F}_c(\mathbf{U})-\mathbf{F}_v(\mathbf{U},\nabla\mathbf{U})\Big)d\mathbf{S}=\int_{\Omega_i}Q_v(\mathbf{U})d\Omega+\oint_{\partial\Omega_i}\mathbf{Q}_s\cdot d\mathbf{S} \qquad \forall\,\Omega_i\in\Gamma \tag{6.3}$$

There are two kinds of finite volume method:

- The *vertex centered* (FVMVC); where the variables are located at the vertices of the mesh.
- The *cell centered* (FVMCC); where the variables are located on the cell of the mesh.

Both strategies have their advantages and drawbacks. There is no shared agreement in the scientific community about which of these two strategy is better. We decided to rely on the cell center formulation because in our opinion it allows a more straightforward implementation of a flow solver, and a clearer treatment of the boundary conditions.
On equation 6.3 the FVMCC makes three approximations:

- It uses the mean value theorem to approximate the volume integral.
- It approximates the cell boundary integral as a sum of constant flux over its faces.
- The flux term $\mathbf{F}_c$ and $\mathbf{F}_v$ are approximated by suitable numerical fluxes $\tilde{\mathbf{F}}_c$ and $\tilde{\mathbf{F}}_v$.

With these approximations equation 6.3 is recasted into:

$$\frac{\partial}{\partial t}\mathbf{U}_i + \frac{1}{\Omega_i}\sum_{faces}\left(\tilde{\mathbf{F}}_c(\mathbf{U}) - \tilde{\mathbf{F}}_v(\mathbf{U},\nabla\mathbf{U})\right)\cdot\mathbf{n} = Q_{v\,i}(\mathbf{U}) + \frac{1}{\Omega_i}\sum_{faces}\mathbf{Q}_s\cdot\mathbf{n}A_{face} \qquad \forall\,\Omega_i\in\Gamma \quad (6.4)$$

It is now possible to define the residual, R_i, of a cell as:

$$\mathbf{R}_i = \frac{1}{\Omega_i}\sum_{faces}\left(\tilde{\mathbf{F}}_c(\mathbf{U}) - \tilde{\mathbf{F}}_v(\mathbf{U},\nabla\mathbf{U})\right)\cdot\mathbf{n} - Q_{v\,i}(\mathbf{U}) - \frac{1}{\Omega_i}\sum_{faces}\mathbf{Q}_s\cdot\mathbf{n}A_{face} \qquad \forall\,\Omega_i\in\Gamma \quad (6.5)$$

This residual defines the space discretization of the FVMCC. Given this term, the associated ordinary differential equation is:

$$\frac{d}{dt}\mathbf{U}_i + \mathbf{R}_i = 0 \qquad \forall\,\Omega_i\in\Gamma \quad (6.6)$$

There have been proposed severeal methods to compute the numerical fluxes $\tilde{\mathbf{F}}_c$ and $\tilde{\mathbf{F}}_v$, and not all of them lead to a reliable solution of the Navier-Stokes Equations. In paragraphs 6.2 and 6.3 we describe how we have decided to compute the convective and the viscous fluxes respectively. Regarding the source term, their computation depends on the phenomena they are representing. In the third part of this thesis we will deal with the computation of the heat source term deriving from the radiation phenomena.
In paragraph 6.4 we describe how the whole solver's machine works, while in paragraph 6.5 we shall address the implementation of some boundary conditions.

6.2 Discretization of the Covective Fluxes

For each face of the mesh we have to compute the convective numerical flux $\tilde{\mathbf{F}}_c$, which, in general, is in function of the values of the conservative variables inside the two cells that share the face. The steps needed to compute this term are:

1. Evaluation of cell centre gradients.

2. Solution Recontruction.
3. Application of a limiter function.
4. Computation of the flux.

In the following subparagraphs we shall describe the above steps.

6.2.1 Solution Reconstruction

In FVMCC variables are broadly supposed to be located on the cell. The interpretation of this statement varies depending on whether we consider *first-order* or *higher-order* approximations. Specifically we have:

1. First-order approximation. Here it is supposed that the variables' values are constant over the whole volume of the cell. We speak of *piecewise constant solution recontruction.*
2. Higher-order approximation. Here it is supposed that the variables' values are located at the centre of the cell, therefore in order to compute their values on a cell's face we have to suitably *recontruct* their values. We speak of *piecewise linear or higher solution reconstruction.*

It appears clear that in general we have to associate to each face what are called the *right state* and the *left state*. The former is the solution reconstructed values on the face of the cell in which the face normal vector is pointing towards. Viceversa, the later is the solution reconstructed values on the face of the cell in which the face normal is pointing away from.
There are different strategies in order to perform a solution reconstruction; they differ according if they work for:

- Structured or also unstructured meshes.
- 2D or also 3D meshes.
- Triangular or mixed (that is, any combination of cell geometry) meshes.
- Computational complexity.

Since each strategy has its own advantages we decided to implement a base class for solution reconstruction and let the concrete classes to derive from it. In this way the concrete finite volume method (see paragraph 4.2) interfaces only with the base class, and we are able to dynamically change the concrete solution reconstruction strategy according to our needs. This capability is accomplished with an RCP object, which is declared as a pointer to the base class but that actually points towards one of the derived classes.
Currently we have implemented three solution reconstruction strategies: which are able to deal with all possible situations; they are:

1. Piecewise constant reconstruction.
2. Piecewise linear recontruction of Barth and Jespersen.
3. Piecewise quadratic recontruction of Barth.

We shall now decribe how they works.

Piecewise Constant Recontruction With this strategy we simply have:

$$\begin{cases} \mathbf{U}_L = \mathbf{U}_i \\ \mathbf{U}_R = \mathbf{U}_j \end{cases} \tag{6.7}$$

Where $\mathbf{U}_L$ and $\mathbf{U}_R$ denotes the left and the right state respectively. Whereas $\mathbf{U}_i$ and $\mathbf{U}_j$ denotes the variable values of the cell at the left and at the right, respectively, of the face. Right and left sides are defined according if they, respectively, are or not in the direction of the face normal vector.

Piecewise Linear Recontruction Barth and Jespersen introduced the piecewise linear recontruction in [25]. With this strategy we compute the right and left state according to:

$$\begin{cases} \mathbf{U}_L = \mathbf{U}_i + \mathbf{\Psi}_i(\nabla \mathbf{U}_i \cdot \mathbf{r}_L) \\ \mathbf{U}_R = \mathbf{U}_j + \mathbf{\Psi}_j(\nabla \mathbf{U}_j \cdot \mathbf{r}_R) \end{cases} \tag{6.8}$$

Where $\mathbf{\Psi}$ is what is called the limiter function, better explained in paragraph 6.2.3. $\nabla \mathbf{U}$ is the gradient of the variables at the centre of the cell, better explained it in paragraph 6.2.2. The vectors $\mathbf{r}_L$ and $\mathbf{r}_R$ point from the cell-centroid to the face-midpoint.
It can be easily shown that this strategy corresponds to a Taylor-series expansion around the neighboring centres of the face, where only the linear term is retained. It can be demonstrated that it is second-order accurate on regular meshes and it performs an exact linear recontruction of the solution if the gradient $\nabla \mathbf{U}$ is evaluated without errors.

Piecewise High-Order Recontruction The second-order is normally the high-order level of choice for finite volume method. Order higher than the second are usually computationally expensive and may arise several mesh-related issues. In [24], [23], [83], and [82], some quadratic solution recontruction strategies have been formulated. We believe they are a very good approach but they can be computationaly involving. Instead in [32] and [31] a more efficient approach has been developed, so that we decided on its implementation. Basically it uses a Taylor series truncated after the quadratic term, so that:

$$\begin{cases} \mathbf{U}_L = \mathbf{U}_i + \mathbf{\Psi}_{i,1}(\nabla \mathbf{U}_i \cdot \mathbf{r}_L) + \frac{1}{2}\mathbf{\Psi}_{i,2}(\mathbf{r}_L^T \mathbf{H}_i \mathbf{r}_L) \\ \mathbf{U}_R = \mathbf{U}_j + \mathbf{\Psi}_{j,1}(\nabla \mathbf{U}_j \cdot \mathbf{r}_R) + \frac{1}{2}\mathbf{\Psi}_{j,2}(\mathbf{r}_R^T \mathbf{H}_j \mathbf{r}_R) \end{cases} \tag{6.9}$$

Where $\mathbf{H}$ denotes the Hessian matrix. The limiters $\mathbf{\Psi}_{i,1}$ and $\mathbf{\Psi}_{i,2}$ regard the first and quadratic term respectively. This reconstruction method is formallly third-order accurate on regular meshes.

6.2.2 Cell Centre Gradients

We have implemented two methods to compute the variables's gradients at the cell's centre, they are:

- Green-Gauss Scheme.
- Least-Square Approach.

We shall now describe both of them.

Green-Gauss Scheme This method approximates the gradient of a scalar variable V inside a control volume as:

$$\nabla V \simeq \frac{1}{\Omega}\int_{\partial\Omega} V\mathbf{n}dS \tag{6.10}$$

In a cell-centred scheme equation 6.10 can be rewritten as:

$$\nabla \mathbf{U}_i \simeq \frac{1}{\Omega}\sum_{faces}\frac{1}{2}(\mathbf{U}_i + \mathbf{U}_j)\mathbf{n}_{ij}A_j \tag{6.11}$$

Where $\nabla\mathbf{U}_i$ is the gradient of the conservative variables inside cell i. $\sum_{faces}$ is a summation extended over all faces j of cell i, so that, $\mathbf{U}_i$ and $\mathbf{U}_j$ are the conservative variables inside cell i and a cell j which is in the other side of face j; $\mathbf{n}_{ij}$ is the normal vector of face j pointing toward cell j, finally A_j denotes the area of face j.
The Green-Gauss scheme is extremely attractive, since it evaluates gradients with a simple formula that is easy to implement and fast to compute. Unfortunately, as demonstrated in [48], it has the drawback that in certain mixed meshes it could compute highly inaccurate gradient in faces where cells of different geometries meet.

Least-Square Approach The least-square approach to compute the gradient in finite volume applications was first introduced in [22]. This formula is based on the following equation:

$$(\nabla\mathbf{U}_i)\cdot\mathbf{r}_{ij} = \mathbf{U}_j - \mathbf{U}_i \tag{6.12}$$

Applaying equation 6.12, to each face of cell i, we obtain a linear system $A\mathbf{x} = \mathbf{b}$; following [21] this system can be solved with a Gram-Schmidt process. The outcome of this process is the following weighted sum:

$$\nabla\mathbf{U}_i = \sum_{faces}\omega_{ij}(\mathbf{U}_j - \mathbf{U}_i) \tag{6.13}$$

Where the weights are defined as:

$$\omega_{ij} = \left\{ \begin{array}{l} \alpha_{ij,1} - \frac{r_{12}}{r_{11}}\alpha_{ij,2} + \beta\alpha_{ij,3} \\ \alpha_{ij,2} - \frac{r_{23}}{r_{22}}\alpha_{ij,3} \\ \alpha_{ij,3} \end{array} \right\} \tag{6.14}$$

The terms in equation 6.13 are given by:

$$\begin{aligned}
\alpha_{ij,1} &= \frac{\Delta x_{ij}}{r_{11}^2} \\
\alpha_{ij,2} &= \frac{1}{r_{22}^2}\left(\Delta y_{ij} - \frac{r_{12}}{r_{11}}\Delta x_{ij}\right) \\
\alpha_{ij,3} &= \frac{1}{r_{33}^2}\left(\Delta z_{ij} - \frac{r_{23}}{r_{22}}\Delta y_{ij} + \beta \Delta x_{ij}\right) \\
\beta &= \frac{r_{12}r_{23} - r_{13}r_{22}}{r_{11}r_{22}}
\end{aligned} \tag{6.15}$$

and:

$$\begin{aligned}
r_{11} &= \sqrt{\textstyle\sum_{faces}(\Delta x_{ij})^2} \\
r_{12} &= \frac{1}{r_{11}} \textstyle\sum_{faces} \Delta x_{ij}\Delta y_{ij} \\
r_{22} &= \sqrt{\textstyle\sum_{faces}(\Delta y_{ij})^2 - r_{12}^2} \\
r_{13} &= \frac{1}{r_{11}} \textstyle\sum_{faces} \Delta x_{ij}\Delta z_{ij} \\
r_{23} &= \frac{1}{r_{22}}\left(\textstyle\sum_{faces} \Delta y_{ij}\Delta z_{ij} - \frac{r_{12}}{r_{11}} \textstyle\sum_{faces} \Delta x_{ij}\Delta z_{ij}\right) \\
r_{33} &= \sqrt{\textstyle\sum_{faces}(\Delta z_{ij})^2 - (r_{13}^2 + r_{23}^2)}
\end{aligned} \tag{6.16}$$

6.2.3 Limiter Function

Second and high-order spatial discretization requires the use of particular functions, called *limiter functions*, which prevent the generation of oscillations and spurious solutions in regions where high variables' gradients are present, such as near shocks and contact discontinuities. Limiter functions attempt to implement the so called monotonicity preserving property, which states that the maxima in the flow field must be non-increasing, minima non-decreasing, and no new local extrema may be created during the time evolution.
There have been developed several limiter functions, each one with its own advantages over the others for certain situations. Like for the case of the solution reconstruction, we have implemented a base class which works as an interface between the concrete derived classes which implement individual limiter functions method and the finite volume solver object. Currently we have implemented two methods which are considered the best solutions for unstructured meshes, they are the limiter of Barth and Jespersen, and the limiter of Venkatakrishnan. We shall now describe both of them.

Limiter of Barth and Jespersen In [25] Barth and Jespersen introduced their limiter function which is defined as below:

$$\mathbf{\Psi}_i = min_j \begin{cases} min\left(1, \frac{\mathbf{U}_{max}-\mathbf{U}_i}{\Delta}\right) & \text{if } \Delta > 0 \\ min\left(1, \frac{\mathbf{U}_{min}-\mathbf{U}_i}{\Delta}\right) & \text{if } \Delta < 0 \\ 1 & \text{if } \Delta = 0 \end{cases} \tag{6.17}$$

Where:

$$\begin{aligned} \Delta &= \nabla \mathbf{U}_i \cdot \mathbf{r}_{ij} \\ \mathbf{U}_{max} &= max(\mathbf{U}_i, max_j \mathbf{U}_j) \\ \mathbf{U}_{min} &= min(\mathbf{U}_i, min_j \mathbf{U}_j) \end{aligned} \tag{6.18}$$

The Barth and Jespersen limiter can be successfully applied for unstructured mixed meshes. Unfortunately it has two issues that in some cases might lead to prevent the convergence to a steady state:

- It smears the discontinuities, it is in fact dissipative.
- It can be activated also in a smooth region due to numerical noise.

Limiter of Venkatakrishnan Venkatakrishnan introduced its limiter in a series of articles: [115], [113], [114], and [116], in which the reliability of this limiter is demonstrated. It is defined as follows:

$$\mathbf{\Psi}_i = min_j \begin{cases} \frac{1}{\Delta}\left(1, \frac{(\Delta_{max}^2+\varepsilon^2)\Delta+2\Delta^2\Delta_{max}}{\Delta_{max}^2+2\Delta^2+\Delta_{max}\Delta+\varepsilon^2}\right) & \text{if } \Delta > 0 \\ \frac{1}{\Delta}\left(1, \frac{(\Delta_{min}^2+\varepsilon^2)\Delta+2\Delta^2\Delta_{min}}{\Delta_{min}^2+2\Delta^2+\Delta_{min}\Delta+\varepsilon^2}\right) & \text{if } \Delta < 0 \\ 1 & \text{if } \Delta = 0 \end{cases} \tag{6.19}$$

Where:

$$\begin{aligned} \Delta_{max} &= \mathbf{U}_{max} - \mathbf{U}_i \\ \Delta_{min} &= \mathbf{U}_{min} - \mathbf{U}_i \end{aligned} \tag{6.20}$$

The parameter ε has the purpose of controlling the "amount of limiting". Setting it to zero means full limiting, but it can halt the convergence. On the contrary a high value of ε can lead to a limiter close to unity, therefore useless. A good compromise is to set it proportional to the local lenght; for example we have set it equal to:

$$\varepsilon = \sqrt{k\Omega_i} \tag{6.21}$$

With k a constant of $\mathcal{O}(1)$.

6.2.4 Computation of the Convective Fluxes

In the framework of Finite Volume Method in order to compute the convective fluxes we have basically four family of schemes:

- *Central.* The basic idea of the central scheme is to compute the convective fluxes at a face of the control volume from the arithmetic average of the conservative variables on both sides of the face.
- *Flux-vector splitting.* This methods can be viewed as the first level of upwind schemes, since they only account for the direction of wave propagation.
- *Flux-difference splitting.* In contrast to the flux-vector splitting schemes, the flux-difference splitting considers not only the direction of wave (information) propagation, but also the waves themselves.
- *Total variation diminishing (TVD).* The TVD schemes are based on a concept aimed at preventing the generation of new extrema in the flow solution. It accomplishes this task by making the total variation of the solution decrease in time.

Each method of the above families is more advantageous than others in specific situations. Therefore we develop an implementation inside which we can dynamically plug and/or switch to any convective flux scheme. The program strategy resembles that used for the implementation of the solution reconstruction and the limiter functions; that is a base class which acts as interface between the concrete derived classes implementing individual convective flux schemes and the finite volume solver object. Currently we have implemented the following schemes:

1. CUSP;
2. AUSM+;
3. AUSM+up;
4. ROE;
5. A modified Roe scheme, let's call it as: All-Mach Roe (*AMRoe*). This is a scheme which we have developed in this thesis and which can operate also at low mach number flows.

We shall now briefly describe all these schemes.

AUSM+

The Advection Upstream Splitting Method family of convective flux scheme was introduced for the first time in [73] and [69], and it was later modified in [117], and finally an improved version called AUSM+ was presented in [71] and [70].
The basic idea behind the AUSM family comes from the observation that the convective flux vector can be splitted in two physically distinct parts: the strictly convective part, $\mathbf{F}_c^{(c)}$, and the pressure part, $\mathbf{F}_c^{(p)}$:

$$\mathbf{F}_c \cdot \mathbf{n} = \begin{pmatrix} \rho u & \rho v & \rho w \\ \rho u^2 + p & \rho uv & \rho uw \\ \rho vu & \rho v^2 + p & \rho vw \\ \rho wu & \rho wv & \rho w^2 + p \\ \rho uh_0 & \rho vh_0 & \rho wh_0 \end{pmatrix} \cdot \mathbf{n} = \underbrace{V \cdot \begin{pmatrix} \rho \\ \rho u \\ \rho v \\ \rho w \\ \rho h_0 \end{pmatrix}}_{\mathbf{F}_c^{(c)}} + \underbrace{\begin{pmatrix} 0 \\ \mathbf{n}_x p \\ \mathbf{n}_y p \\ \mathbf{n}_z p \\ 0 \end{pmatrix}}_{\mathbf{F}_c^{(p)}} \tag{6.22}$$

Where V denotes the contravariant velocity, that is, the scalar product between the velocity vector and the face normal vector $\mathbf{n}$. The convective part is governed by the convective wave whereas the pressure term is governed by the acoustic wave speed. Therefore $\mathbf{F}_c^{(c)}$ is discretized, in both supersonic and subsonic cases, in purely upwind manner by taking either the left or the right state ($\mathbf{U}_L$ and $\mathbf{U}_R$ respectively), depending on the sign of V. Conversely $\mathbf{F}_c^{(p)}$ is discretized in upwind manner only in the supersonic case, while the the subsonic case is discretized centered manner.
Now it is a matter of managing to translate the described AUSM strategy into formulas, we shall now do it following [71]. Let's start computing the advective Mach numbers of the left and right states:

$$M_L = \frac{V_L}{a} \qquad \text{and} \qquad M_R = \frac{V_R}{a} \tag{6.23}$$

Where V_L and V_R are the contravariant velocity computed based on the left and the right state respectively. The sound speed over a face a is computed as:

$$a = min(\tilde{a}_L, \tilde{a}_R) \tag{6.24}$$

With:

$$\tilde{a}_L = \frac{\hat{a}_L^2}{max(\hat{a}_L, |V_L|)} \qquad \text{and} \qquad \tilde{a}_R = \frac{\hat{a}_R^2}{max(\hat{a}_R, |V_R|)} \tag{6.25}$$

In equations 6.25 $\hat{a}$ denotes the critical speed of sound:

$$\hat{a} = \sqrt{\frac{2(\gamma - 1)}{\gamma + 1} h_0} \tag{6.26}$$

In which a_t is the speed of sound based on the total entalpy h_0. We now define the convective speed on the face, m, as:

$$m = \mathcal{M}_{(4)}^{+}(M_L) + \mathcal{M}_{(4)}^{-}(M_R) \tag{6.27}$$

Where:

$$\mathcal{M}_{(1)}^{\pm}(M) = \frac{1}{2}(M \pm |M|) \tag{6.28}$$

$$\mathcal{M}_{(2)}^{\pm}(M) = \pm\frac{1}{4}(M \pm 1)^2 \tag{6.29}$$

$$\mathcal{M}_{(4)}^{\pm}(M) = \begin{cases} \mathcal{M}_{(1)}^{\pm}(M) & \text{if } |M| \geq 1 \\ \mathcal{M}_{(2)}^{\pm}(M)(1 \mp 16\beta\mathcal{M}_{(2)}^{\mp}(M)) & \text{otherwise} \end{cases} \tag{6.30}$$

With:

$$-\frac{1}{16} \leq \beta \leq \frac{1}{2} \tag{6.31}$$

After this we define:

$$m^{\pm} = \frac{1}{2}(m \pm |m|) \tag{6.32}$$

Regarding the pressure term we define the numerical pressure, $\tilde{p}$:

$$\tilde{p} = \mathcal{P}^{+}(M_L)p_L + \mathcal{P}^{-}(M_R)p_R \tag{6.33}$$

Where:

$$\mathcal{P}^{\pm}(M) = \begin{cases} \frac{1}{M}\mathcal{M}^{\pm}_{(1)}(M) & \text{if } |M| \geq 1 \\ \mathcal{M}^{\pm}_{(2)}(M)(\pm 2 - M \mp 16\alpha M \mathcal{M}^{\mp}_{(2)}(M)) & \text{otherwise} \end{cases} \tag{6.34}$$

With:

$$-\frac{3}{16} \leq \alpha \leq \frac{1}{8} \tag{6.35}$$

Finally the numerical convective flux, $\tilde{\mathbf{F}}_c$, is computed as:

$$\tilde{\mathbf{F}}_c \cdot \mathbf{n} = a\, m^{+} \begin{pmatrix} \rho \\ \rho u \\ \rho v \\ \rho w \\ \rho h_0 \end{pmatrix}_L + a\, m^{-} \begin{pmatrix} \rho \\ \rho u \\ \rho v \\ \rho w \\ \rho h_0 \end{pmatrix}_R + \begin{pmatrix} 0 \\ \mathbf{n}_x \tilde{p} \\ \mathbf{n}_y \tilde{p} \\ \mathbf{n}_z \tilde{p} \\ 0 \end{pmatrix} \tag{6.36}$$

AUSM+up

In [72] the AUSM+ scheme is extended to function at low Mach numbers, that is in the limit of small value of Mach number. According to [72] AUSM+up is able to solve flow fields at all speed regimes without preconditioning. Following the new formulation a pressure diffusion term is integrated in equation 6.27:

$$m = \mathcal{M}^{+}_{(4)}(M_L) + \mathcal{M}^{-}_{(4)}(M_R) - \frac{K_p}{f_a} max(1 - \sigma \bar{M}^2, 0) \frac{p_R - p_L}{\frac{1}{2}(\rho_R + \rho_L)a^2} \tag{6.37}$$

Where:

$$\bar{M}^2 = \frac{V_R^2 + V_L^2}{2a^2} \tag{6.38}$$

$$f_a(M_0) = M_0(2 - M_0) \tag{6.39}$$

$$M_0^2 = min(1, max(\bar{M}^2, M_\infty^2)) \tag{6.40}$$

$$0 \leq K_p \leq 1 \qquad \sigma \leq 1 \tag{6.41}$$

The pressure flux 6.33 is also modified as follows:

$$\tilde{p} = \mathcal{P}^{+}(M_L)p_L + \mathcal{P}^{-}(M_R)p_R - K_u \mathcal{P}^{+}(M_L)\mathcal{P}^{-}(M_R)(\rho_R + \rho_L) f_a a (V_R - V_L) \tag{6.42}$$

With:

$$\alpha = \frac{3}{16}(-4 + 5f_a^2) \qquad \beta = \frac{1}{8} \qquad 0 \leq K_u \leq 1 \tag{6.43}$$

CUSP

The CUSP scheme [52] computes the face convective flux as:

$$\mathbf{F}_c \cdot \mathbf{n} = \frac{1}{2} V_R \begin{pmatrix} \rho \\ \rho u \\ \rho v \\ \rho w \\ \rho h_0 \end{pmatrix}_R + \frac{1}{2} V_L \begin{pmatrix} \rho \\ \rho u \\ \rho v \\ \rho w \\ \rho h_0 \end{pmatrix}_L + \begin{pmatrix} 0 \\ \mathbf{n}_x \tilde{p} \\ \mathbf{n}_y \tilde{p} \\ \mathbf{n}_z \tilde{p} \\ 0 \end{pmatrix} + \mathbf{d} \tag{6.44}$$

Where $\tilde{p} = \frac{1}{2}(p_R + p_L)$ and $\mathbf{d}$ is an artificial diffusive term introduced in order to stabilize the scheme; it is defined as:

$$\mathbf{d} = \frac{1}{2}\tilde{\alpha}a \left\{ \begin{pmatrix} \rho \\ \rho u \\ \rho v \\ \rho w \\ \rho e_0 \end{pmatrix}_R - \begin{pmatrix} \rho \\ \rho u \\ \rho v \\ \rho w \\ \rho e_0 \end{pmatrix}_L \right\} + \frac{1}{2}\beta a \left\{ \begin{pmatrix} V_R\, \rho \\ V_R\, \rho u + \mathbf{n}_x p \\ V_R\, \rho v + \mathbf{n}_y p \\ V_R\, \rho w + \mathbf{n}_z p \\ V_R\, \rho h_0 \end{pmatrix}_R - \begin{pmatrix} V_L\, \rho \\ V_L\, \rho u + \mathbf{n}_x p \\ V_L\, \rho v + \mathbf{n}_y p \\ V_L\, \rho w + \mathbf{n}_z p \\ V_L\, \rho h_0 \end{pmatrix}_L \right\} \tag{6.45}$$

With:

$$\tilde{\alpha}a = \alpha a - \beta \frac{1}{2}(V_R + V_L) \tag{6.46}$$

$$\alpha a = |M| \tag{6.47}$$

$$\beta = \begin{cases} max(0, 2M - 1) & \text{if } 0 \leq M \leq 1 \\ min(0, 2M + 1) & \text{if } -1 \leq M \leq 0 \end{cases} \tag{6.48}$$

Near a stagnation point α may be modified to:

$$\alpha = \frac{1}{2}\left(\alpha_0 + \frac{|M|^2}{\alpha_0}\right) \qquad if\ |M| \leq \alpha_0 \tag{6.49}$$

Roe

Roe's flux splitting scheme [96] is based on the approximate solution of a Riemann problem: it decomposes the flux difference over a face of the control volume into a sum of wave contributions while ensuring the conservation properties of the Euler equations. With its variants it is considered the most accurate scheme among the finite volume based discretization techniques for convective fluxes. Roe scheme computes the convective flux on a face, $\mathbf{F}^{Roe}$, according to the following expression:

$$\mathbf{F}^{Roe} = \frac{1}{2}(\mathbf{F}_n(\mathbf{U}_R) + \mathbf{F}_n(\mathbf{U}_L)) - \frac{1}{2}|\mathbf{A}^{Roe}|(\mathbf{U}_R - \mathbf{U}_L) \tag{6.50}$$

The Roe matrix, $\mathbf{A}^{Roe}$, is identical to the convective flux Jacobian where the flow variables are replaced by the so called Roe-averaged variables:

$$\mathbf{A}^{Roe} = \frac{\partial \mathbf{F}_n}{\partial \mathbf{U}} \tag{6.51}$$

$$\tilde{\rho} = \sqrt{\rho_R \rho_L} \tag{6.52}$$

$$\tilde{u} = \frac{u_R\sqrt{\rho_R} + u_L\sqrt{\rho_L}}{\sqrt{\rho_R} + \sqrt{\rho_L}} \tag{6.53}$$

$$\tilde{v} = \frac{v_R\sqrt{\rho_R} + v_L\sqrt{\rho_L}}{\sqrt{\rho_R} + \sqrt{\rho_L}} \tag{6.54}$$

$$\tilde{w} = \frac{w_R\sqrt{\rho_R} + w_L\sqrt{\rho_L}}{\sqrt{\rho_R} + \sqrt{\rho_L}} \tag{6.55}$$

$$\tilde{h}_0 = \frac{h_{0\,R}\sqrt{\rho_R} + h_{0\,L}\sqrt{\rho_L}}{\sqrt{\rho_R} + \sqrt{\rho_L}} \tag{6.56}$$

$$\tilde{V} = \tilde{u}\mathbf{n}_x + \tilde{v}\mathbf{n}_y + \tilde{w}\mathbf{n}_z \tag{6.57}$$

$$\tilde{q}^2 = \tilde{u}^2 + \tilde{v}^2 + \tilde{w}^2 \tag{6.58}$$

$$\tilde{a} = \sqrt{(\gamma - 1)\left(\tilde{h}_0 - \frac{1}{2}\tilde{q}^2\right)} \tag{6.59}$$

After some algebraic manipulations we arrive to:

$$|\mathbf{A}^{Roe}|(\mathbf{U}_R - \mathbf{U}_L) = |\Delta\mathbf{F}_1| + |\Delta\mathbf{F}_{2,3,4}| + |\Delta\mathbf{F}_5| \tag{6.60}$$

With:

$$|\Delta\mathbf{F}_1| = |\tilde{V} - \tilde{a}|\left(\frac{\Delta p - \tilde{\rho}\tilde{a}\Delta V}{2\tilde{a}^2}\right)\begin{pmatrix} 1 \\ \tilde{u} - \tilde{a}\mathbf{n}_x \\ \tilde{v} - \tilde{a}\mathbf{n}_y \\ \tilde{w} - \tilde{a}\mathbf{n}_z \\ \tilde{h}_0 - \tilde{a}\tilde{V} \end{pmatrix} \tag{6.61}$$

$$|\Delta\mathbf{F}_{2,3,4}| = |\tilde{V}|\left\{\left(\Delta\rho - \frac{\Delta p}{\tilde{a}^2}\right)\begin{pmatrix} 1 \\ \tilde{u} \\ \tilde{v} \\ \tilde{w} \\ \frac{1}{2}\tilde{q}^2 \end{pmatrix} + \tilde{\rho}\begin{pmatrix} 0 \\ \Delta u - \Delta V\mathbf{n}_x \\ \Delta v - \Delta V\mathbf{n}_y \\ \Delta w - \Delta V\mathbf{n}_z \\ \tilde{u}\Delta u + \tilde{v}\Delta v + \tilde{w}\Delta w - \tilde{V}\Delta V \end{pmatrix}\right\} \tag{6.62}$$

$$|\Delta\mathbf{F}_5| = |\tilde{V} + \tilde{a}|\left(\frac{\Delta p + \tilde{\rho}\tilde{a}\Delta V}{2\tilde{a}^2}\right)\begin{pmatrix} 1 \\ \tilde{u} + \tilde{a}\mathbf{n}_x \\ \tilde{v} + \tilde{a}\mathbf{n}_y \\ \tilde{w} + \tilde{a}\mathbf{n}_z \\ \tilde{h}_0 + \tilde{a}\tilde{V} \end{pmatrix} \tag{6.63}$$

In equations 6.61, 6.62 and 6.63 the terms $|\tilde{V} - \tilde{a}|$, $|\tilde{V}|$ and $|\tilde{V} + \tilde{a}|$ are the eigenvalues, Λ, of $\mathbf{A}^{Roe}$ and they represent the velocities with which the information travels inside the flow domain.
The Roe scheme is notoriously affected by the so-called "carbuncle phenomenon", where a perturbation grows ahead of a strong bow shock along the stagnation line, [49]. The underlying difficulty is that the original scheme does not recognise the sonic point. In order to solve this problem, the modulus of the eigenvalues can be modified using Harten's entropy correction:

$$|\Lambda| = \begin{cases} |\Lambda| & \text{if } |\Lambda| \geq \delta \\ \frac{\Lambda^2+\delta^2}{2\delta} & \text{if } |\Lambda| < \delta \end{cases} \tag{6.64}$$

Where δ can be set equal to some fraction of the sound speed: $\delta \simeq 0.1a$.

Modified Roe (AMRoe)

It is well known that the Roe schemes do not perform well on low value of Mach number, indeed, for nearly-incompressible flows it produces unphysical discrete results. Here we want to propose a simple modification to the Roe scheme which allows to obtain physical results also for low Mach number flow regimes. We will not explain the general issues involved in extending a density-based method to incompressible flows simulations, the interested reader is referred to [50] and [39].
Recently, several articles have been published proposing different solution to fix the Roe scheme at low Mach numbers: [33], [36], [106], [66], [67], [109], [118] and [46]. We tried to test the solutions proposed in those articles, but unfortunatelly among them we found no one able to properly fix both the checkerboard and the accuracy problems. Therefore, we investigated on our own the possibility for a fix, possibly, simple to implement and able to produce accurate and checkerboard free simulations. Our fix has two ingredients:

- Scale the ΔV variable with a blending function f_m. Indeed, guided by our intuition and some numerical experiments, we discovered that ΔV is the key term to be manipulated in order to extend the Roe scheme to all flow regimes. Fundamentally, ΔV needs to be suitably scaled.
- Scaling the eigenvalues at low Mach number with a blending function f_n. In order to prevent the linear waves from disappearing for $\tilde{V} \to 0$, like at stagnation points and for grid aligned flow.

Basically our modified Roe convective flux is computed as:

$$\mathbf{F}_c \cdot \mathbf{n} = \frac{1}{2} V_R \begin{pmatrix} \rho \\ \rho u \\ \rho v \\ \rho w \\ \rho h_0 \end{pmatrix}_R + \frac{1}{2} V_L \begin{pmatrix} \rho \\ \rho u \\ \rho v \\ \rho w \\ \rho h_0 \end{pmatrix}_L + \begin{pmatrix} 0 \\ \mathbf{n}_x \hat{p} \\ \mathbf{n}_y \hat{p} \\ \mathbf{n}_z \hat{p} \\ 0 \end{pmatrix} + |\widehat{\Delta \mathbf{F}_1}| + |\widehat{\Delta \mathbf{F}_{2,3,4}}| + |\widehat{\Delta \mathbf{F}_5}| \tag{6.65}$$

Where:

$$|\widehat{\Delta \mathbf{F}_1}| = |\widehat{\Lambda_1}| \left(\frac{\Delta p - \tilde{\rho}\tilde{a}\widehat{\Delta V}}{2\tilde{a}^2} \right) \begin{pmatrix} 1 \\ \tilde{u} - \tilde{a}\mathbf{n}_x \\ \tilde{v} - \tilde{a}\mathbf{n}_y \\ \tilde{w} - \tilde{a}\mathbf{n}_z \\ \tilde{h}_0 - \tilde{a}\tilde{V} \end{pmatrix} \tag{6.66}$$

$$|\widehat{\Delta \mathbf{F}_{2,3,4}}| = |\widehat{\Lambda_{2,3,4}}| \left\{ \left(\Delta\rho - \frac{\Delta p}{\tilde{a}^2} \right) \begin{pmatrix} 1 \\ \tilde{u} \\ \tilde{v} \\ \tilde{w} \\ \frac{1}{2}\tilde{q}^2 \end{pmatrix} + \tilde{\rho} \begin{pmatrix} 0 \\ \Delta u - \widehat{\Delta V}\mathbf{n}_x \\ \Delta v - \widehat{\Delta V}\mathbf{n}_y \\ \Delta w - \widehat{\Delta V}\mathbf{n}_z \\ \tilde{u}\Delta u + \tilde{v}\Delta v + \tilde{w}\Delta w - \tilde{V}\widehat{\Delta V} \end{pmatrix} \right\} \tag{6.67}$$

$$|\widehat{\Delta \mathbf{F}_5}| = |\widehat{\Lambda_5}| \left(\frac{\Delta p + \tilde{\rho}\tilde{a}\widehat{\Delta V}}{2\tilde{a}^2} \right) \begin{pmatrix} 1 \\ \tilde{u} + \tilde{a}\mathbf{n}_x \\ \tilde{v} + \tilde{a}\mathbf{n}_y \\ \tilde{w} + \tilde{a}\mathbf{n}_z \\ \tilde{h}_0 + \tilde{a}\tilde{V} \end{pmatrix} \tag{6.68}$$

With the following definitions:

$$|\widehat{\Lambda_1}| = |\tilde{V} - \tilde{a}|; \qquad |\widehat{\Lambda_{2,3,4}}| = |\tilde{V}|; \qquad |\widehat{\Lambda_5}| = |\tilde{V} + \tilde{a}| \tag{6.69}$$

$$|\widehat{\Lambda_i}| = \begin{cases} |\widehat{\Lambda_i}| & \text{if } |\widehat{\Lambda_i}| \geq \epsilon \\ \frac{\widehat{\Lambda_i}^2 + \epsilon^2}{2\epsilon} & \text{if } |\widehat{\Lambda_i}| < \epsilon \end{cases} \tag{6.70}$$

$$\epsilon = \xi_a \cdot f_n \cdot \tilde{a} + \xi_b \tag{6.71}$$

$$f_n = tanh\left(\frac{|M_n|}{\xi_c} \right) \tag{6.72}$$

$$M_n = \frac{\tilde{V}}{\tilde{a}} \tag{6.73}$$

$$\widehat{\Delta V} = f_m \cdot (V_R - V_L) \tag{6.74}$$

$$f_m = tanh\left(\frac{|M_m|}{\xi_d} + \xi_e |M_{n\,R} - M_{n\,L}| \right) \tag{6.75}$$

$$M_m = \frac{\tilde{V}_m}{\tilde{c}} \tag{6.76}$$

$$M_{n\,R} = \frac{\tilde{V}_{n\,R}}{\tilde{c}} \tag{6.77}$$

$$M_{n\,L} = \frac{\tilde{V}_{n\,L}}{\tilde{c}} \tag{6.78}$$

$$\tilde{V}_m = \sqrt{\tilde{u}^2 + \tilde{v}^2 + \tilde{w}^2} \tag{6.79}$$

$$\tilde{V}_{n\,R} = u_R \mathbf{n}_x + v_R \mathbf{n}_y + w_R \mathbf{n}_z \tag{6.80}$$

$$\tilde{V}_{n\,L} = u_L \mathbf{n}_x + v_L \mathbf{n}_y + w_L \mathbf{n}_z \tag{6.81}$$

$$\hat{p} = \frac{1}{2}(p_R + p_L) \tag{6.82}$$

Where ξ_a, ξ_b, ξ_c, ξ_d and ξ_e are tunable parameters; in all our computations we have set them equal to:

$$\xi_a = 0.1; \qquad \xi_b = 0.1; \qquad \xi_c = 0.15; \qquad \xi_d = 0.3; \qquad \xi_e = 1.0 \tag{6.83}$$

In eq. 6.75 the term: $\frac{|M_m|}{\xi_d}$, has the purpose of decreasing the jump in the normal velocity, $V_R - V_L$, based on the Mach number M_m, which is computed with the interface velocity, $\tilde{V}_m$, not with the normal velocity, $\tilde{V}$. Instead the term $\xi_e |M_{n\,R} - M_{n\,L}|$ has the purpose of reduce the effect of $\frac{|M_m|}{\xi_d}$ in presence of local discontinuities. In this way, it happens that in presence of a shock the AMRoe scheme locally gives raise to the Roe scheme. We want to stress that all the variables involved in all the expressions used in the AMRoe scheme are local variables, it means that AMRoe does not use global variables, and this is an obvious advantage. As we will show in chapter 7, our AMRoe is able to work at all speed regimes, from very low Mach number to supersonic flow regimes.

6.3 Discretization of the Diffusive Fluxes

The diffusive or viscous fluxes have an elliptic nature, therefore all the quantities needed to compute this flux over a face are computed simply by averaging the values that they assume inside the two cells sharing the face. The vector of conservative variable used to compute the viscous flux is thus:

$$\mathbf{U}_{ij} = \frac{1}{2}(\mathbf{U}_i + \mathbf{U}_j) \tag{6.84}$$

Where $\mathbf{U}_{ij}$ is the vector of conservative variables over the face that divide cell i and j. The only task that remains to be defined is the computation of the gradients' values over a face. Since we have already computed the gradients at the centre of the cells in order to compute the convective fluxes, it would be tempting to evaluate the gradient at the face-midpoint by the simple average:

$$\nabla\mathbf{U}_{ij} = \frac{1}{2}(\nabla\mathbf{U}_i + \nabla\mathbf{U}_j) \tag{6.85}$$

Unfortunately as pointed out in [79] this simple strategy have several drawbacks. Nevertheless in [48] a workaround has been defined with the following expression:

$$\nabla\mathbf{U}_{ij} = \overline{\nabla\mathbf{U}}_{ij} - \left[\overline{\nabla\mathbf{U}}_{ij} \cdot \mathbf{t}_{ij} - \frac{\mathbf{U}_j - \mathbf{U}_i}{\|\mathbf{r}_{ij}\|_2}\right]\mathbf{t}_{ij} \tag{6.86}$$

Where:

$$\overline{\nabla\mathbf{U}}_{ij} = \frac{1}{2}(\nabla\mathbf{U}_i + \nabla\mathbf{U}_j) \tag{6.87}$$

and

$$\mathbf{t}_{ij} = \frac{\mathbf{r}_{ij}}{\|\mathbf{r}_{ij}\|_2} \tag{6.88}$$

$\mathbf{r}_{ij}$ is the vector pointing from the cell centroid j to the cell centroid i.

6.4 The Fluid Flow Solver

We implemented a solver for the Euler Equations, and for the Laminar Navier-Stokes Equations. These implementations make full use of the Multiphysics Framework, described in the first part of this thesis, through specific settings; these settings regard mainly the connectivity type and the number of overlap region layers. Regarding the former we have set the IMesh object to have the following connectivities:

- cell-to-face,
- cell-to-vertex,
- face-to-cell,
- face-to-vertex.

We have prescribed only one layer of overlap region for all cases but with the quadratic solution reconstruction.
The computation of the residual vector is accomplished with two loops, one over the cells for the computation of volume source terms, and one over the faces for the computation of convective flux term, diffusive flux term (this only for the Navier-Stokes equations), and face source term. In each iteration of the loop we add some values to the residual vector. In order to easily and transparently accomplish this task the algorithm makes considerable use of the iterator objects (see paragraph 3.5).
All settings regarding:

- solution recontruction method;
- method to compute variables' gradients;
- numerical convective flux scheme;
- limiter functions;

and any other possible choices can be directly set by the user through the input XML file.

6.5 The Boundary Conditions

Several boundary conditions have been implemented, among others we cite:

- Supersonic Inlet,
- Supersonic Outlet,
- Subsonic Inlet,
- Subsonic Outlet,
- Inviscid Wall,
- Viscous Adiabatic Wall,
- Viscous Constant Temperatre Wall.

The purpose of computing is insight, not numbers.

Richard Hamming

Chapter 7

Fluid Flow Solver Validation

7.1 Context

We will consider several test cases in order to validate the finite volume solver to a variety of flow conditions. The first test case is the inviscid subsonic flow past a cylinder. We will compare the numerical result with the solution obtained with the potential flow model. The second case will be typical of internal channel flows. The next case will be representative of supersonic flow with an oblique shock. The last case is the computation of the boundary layer over a flat plate. All the cases will be treated assuming perfect gas relations for the considered fluid.
All the simulations shown below are performed with 4 processors. Since the Hydra code is 3D, all the meshes used for the test cases are 3D but with the trick of using: a constant thickness in the z direction and symmetry boundary conditions on the 3D faces parallel to the z coordinate. Moreover all simulation are made with implicit method using the Automatic Differentiation strategy in order to compute the residual Jacobian. The inviscid fluxes are always computed using the modified Roe fluxes strategy which we have developed 6.2.4. The following simulations are therefore intended in order to validate:

- the whole machine of the HYDRA framework;
- the automatic differentiation strategy;
- our modified Roe scheme: *AMRoe*.

Looking at the following results we can state that that the validation is accomplished.

7.2 Cylinder testcase

We have considered a circular mesh with 32 points in the radial direction and 128 points in the circumferential direction. In the inlet boundary we have assumed atmospheric conditions, with pressure = 101300 Pa, Temperature = 288 K, and Mach number = 0.03. With this very low Mach number the flow can be considered as nearly-incompressible. The atmospheric pressure

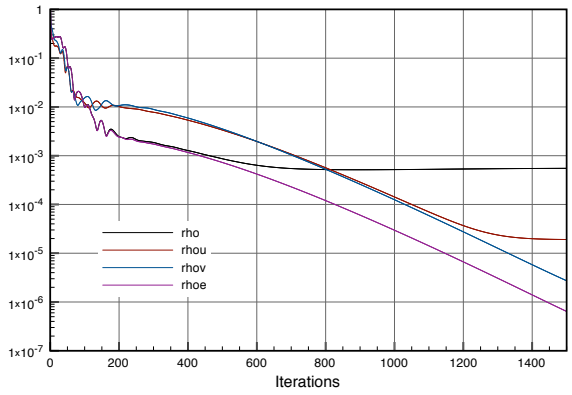
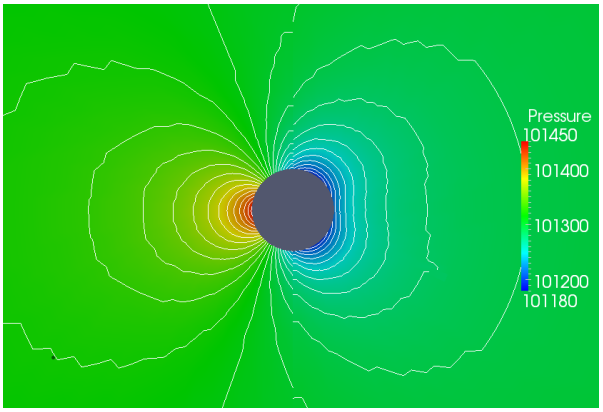


Figure 7.1: Cylinder testcase at inlet Mach 0.03, with the Roe scheme. Convergence history: L_2 norm of the residuals.

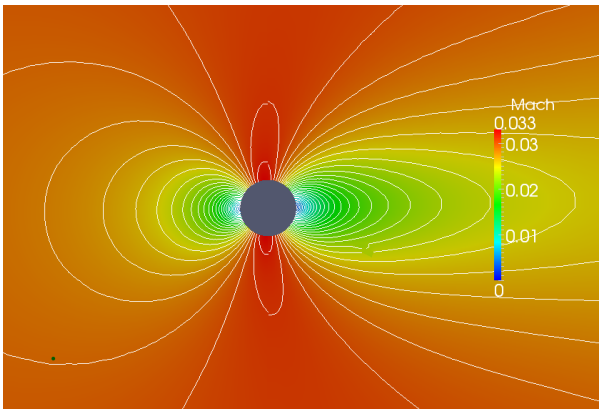
is also imposed at the outlet. Since the computational domain is circular, we will consider that the left half circle of the outer boundary forms the inlet section, while the right half circle will form the outlet section of the computational domain. On the cylinder wall the slip wall boundary condition is imposed. We have used a CFL equal to 25. First of all we show the simulation made with the Roe scheme and afterward the simulation with our modified Roe scheme, 6.2.4.
Regarding the Roe scheme, in fig. 7.1 is shown the convergence history in the L_2 norm of the residuals; in fig. 7.2 are shown the pressure and the Mach contours. Finally in in fig 7.3 we can observe the comparison between the Euler solution and the potential flow solution: the pressure coefficient, C_p, is plotted against the circumferential position, in radiants, on the cylinder's wall. We see a totally disagreement between the two, it means that the Roe scheme is giving a totally unphysical result.

Instead making the simulation with our modified Roe scheme, with the CFL again equal to 25, we finally get a physical result. In fig. 7.4 is shown the convergence history in the L_2 norm of the residuals. We can notice that the residual of the energy decrease rapidly under 1e-7, this is in contrast with other scheme, like AUSM+, which for very low mach number have some difficulties to make the residual decrese under 1e-4. In fig. 7.5 are shown the pressure and Mach contours. We can observe that there is not the checkerboard problem in the solution, it means that our modified Roe scheme is able to properly couple the pressure and the velocity fields. Moreover their aspects reflects what one can expect from an Euler Solver, that is, something very close to the potential flow model solution but with the addition of certain amount of numerical dissipation. This phenomenon is particularly clear looking at the back face of the cylinder where a wake appears. In fig 7.10 again we can observe the comparison between the Euler solution and the potential flow solution: we see a good agreement between the two in the front of the cylinder, whereas in the back part there is an expected lack of matching which can be attributed to:

- Some numerical dissipation. But with a better tuning of the parameters ξ can be quite reduced.
- The coarseness of the mesh. We indeed found that the matching can be significantly



(a)



(b)

Figure 7.2: Cylinder testcase at inlet Mach 0.03, with the Roe scheme, at inlet Mach 0.03. Pressure (a) and Mach (b) contours.

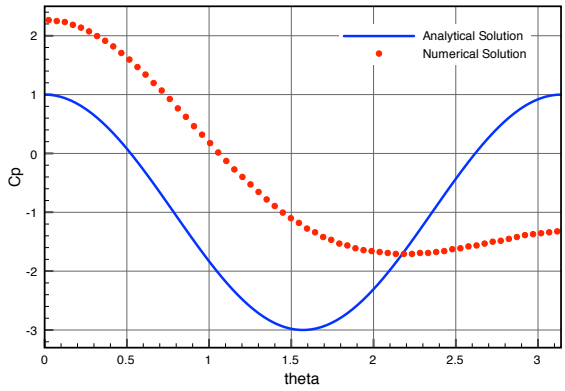


Figure 7.3: Cylinder testcase at inlet Mach 0.03, with the Roe scheme. Pressure coefficient along the cylinder's wall: comparison between the analytical and the numerical solution.

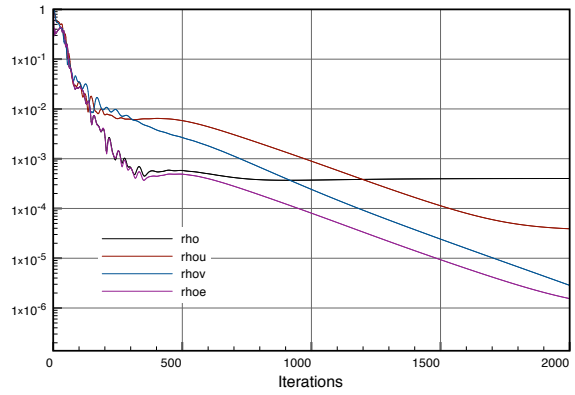


Figure 7.4: Cylinder testcase at inlet Mach 0.03, with our modified Roe scheme. Convergence history: L_2 norm of the residuals.

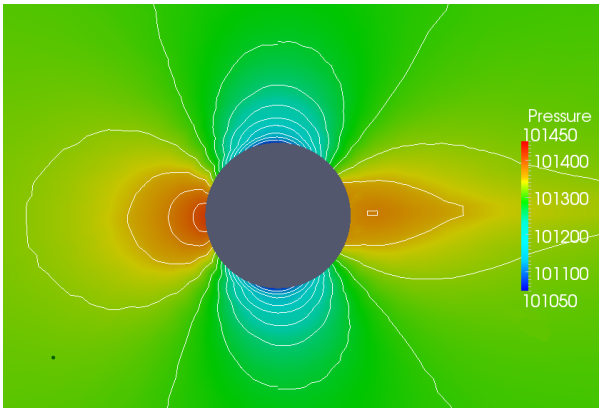
increased with a finer mesh.

We report now the result with the same CFL and Mach number that we obtained with the AUSM+up scheme. Again in fig. 7.7 shows the residuals' convergence; instead fig. 7.8 shows the pressure and Mach contour; finally fig. 7.9 shows the comparison between the analytical and the numerical solution of the pressure coefficient along the cylinder's wall. We can observe that the scheme easily converge, but the solution that we get is not very accurate. In fact in fig. 7.9 we do not observe a good matching while conversely, as we saw in fig. 7.10, our modified Roe scheme allows a quite better agreement.

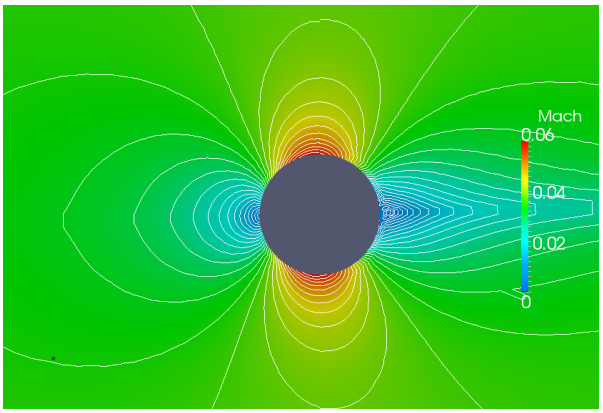
We have tested the modified Roe scheme at several speed regimes, and we have always found good results. As an example, in fig. 7.10 we show the results of a simulation with an inlet Mach number equal to 1.176.

7.3 Channel with a bump testcase

This test case is a representative example of an internal flow configuration. The computational domain consists of a channel of height L and length 3L with, along the bottom wall, a circular arc of length L and thickness equal to 0.1L.
For the inlet we have assumed atmospheric conditions, for pressure and temperature with pressure = 101300 Pa, Temperature = 288 K, and Mach number = 0.588 as incoming flow conditions. The atmospheric pressure is also imposed at the outlet. The CFL is equal to 20.
In fig. 7.11 is shown the convergence history in the L_2 norm of the residuals. In fig. 7.12 are shown the pressure and Mach contours; their aspects reflects what one can expect from an Euler Solver.// In fig. 7.13 are shown the results for the same testcase but with the inlet Mach number equal to 0.7. In this case we can observe the rise of a shock in the back part of the bump. Both the simulations are in good agreement to what we can expect.



(a)



(b)

Figure 7.5: Cylinder testcase at inlet Mach 0.03, with our modified Roe scheme. Pressure (a) and Mach (b) contours.

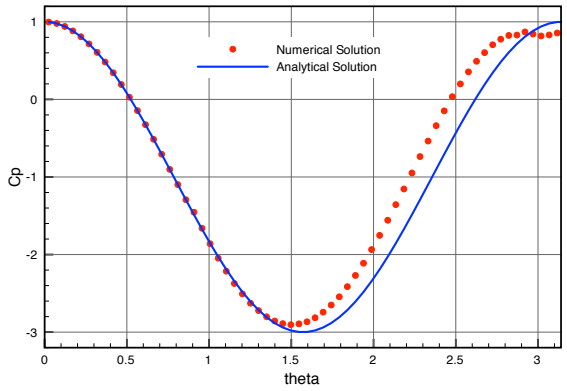


Figure 7.6: Cylinder testcase at inlet Mach 0.03, with our modified Roe scheme. Pressure coefficient along the cylinder's wall: comparison between the analytical and the numerical solution.

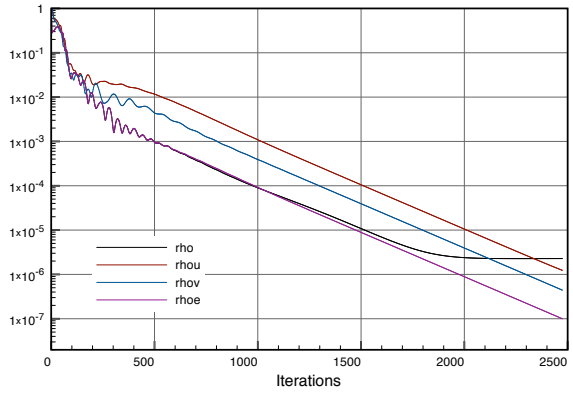
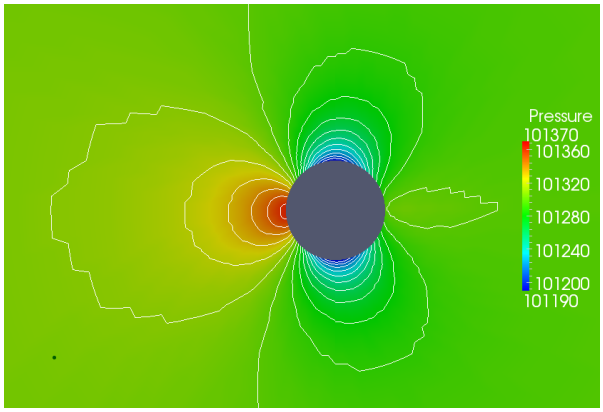
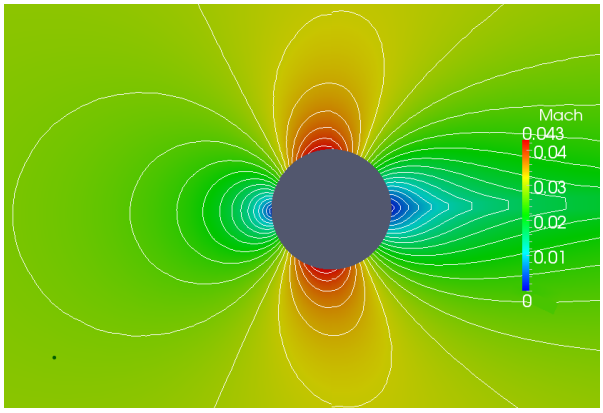


Figure 7.7: Cylinder testcase at inlet Mach 0.03, with AUSM+up scheme. Convergence history: L_2 norm of the residuals.



(a)



(b)

Figure 7.8: Cylinder testcase at inlet Mach 0.03, with AUSM+up scheme. Pressure (a) and Mach (b) contours.

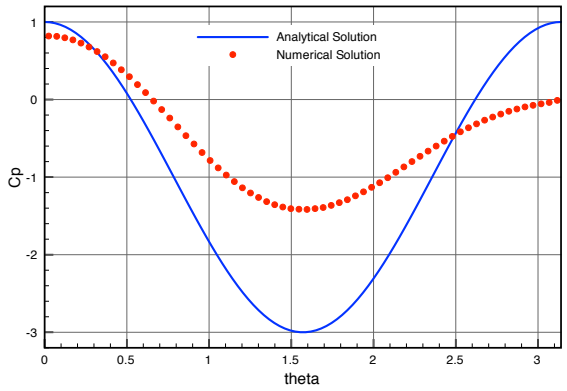
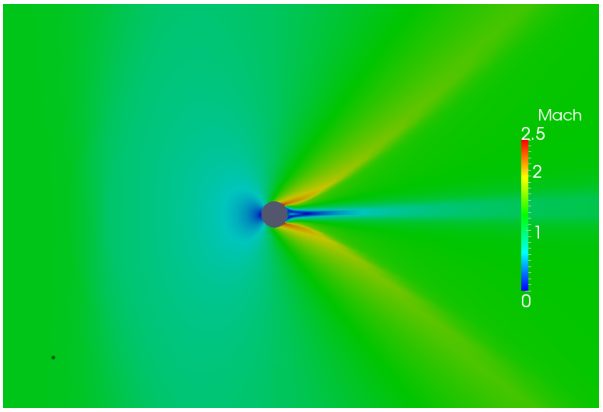
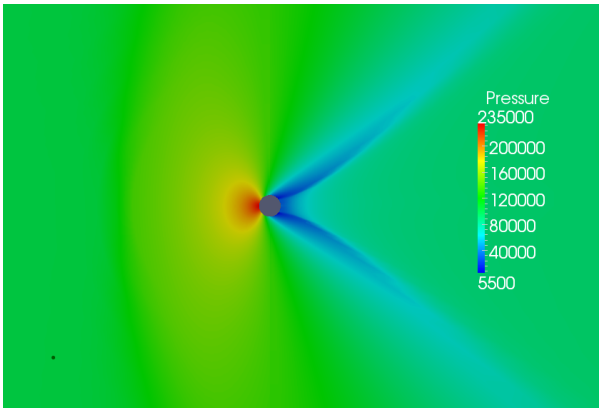


Figure 7.9: Cylinder testcase at inlet Mach 0.03, with AUSM+up scheme. Pressure coefficient along the cylinder's wall: comparison between the analytical and the numerical solution.



(a)



(b)

Figure 7.10: Cylinder testcase at inlet Mach 1.176, with our modified Roe scheme. Pressure (a) and Mach (b) contours.

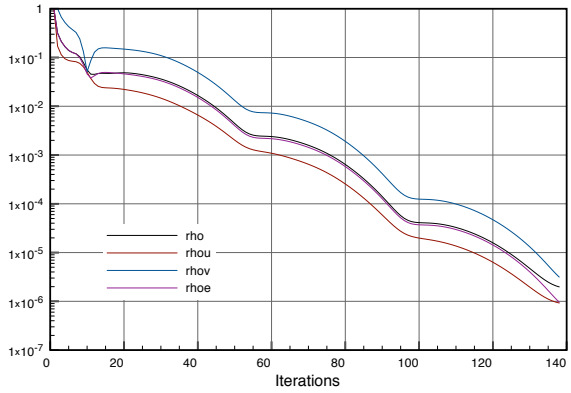
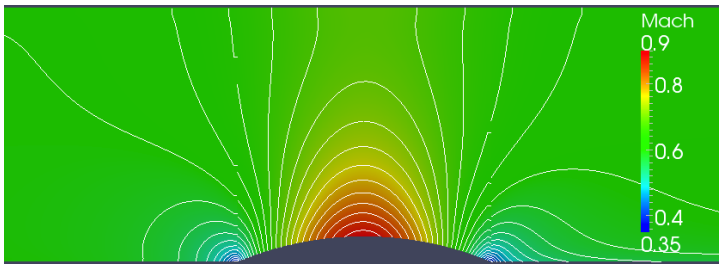
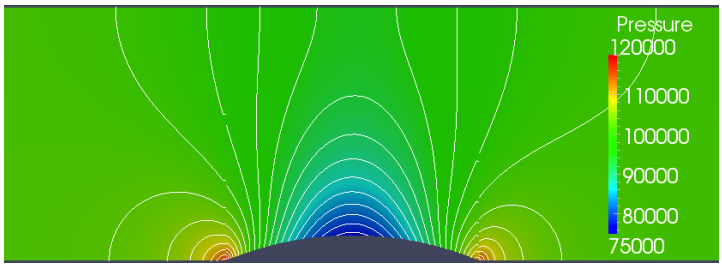


Figure 7.11: Channel testcase, with our modified Roe scheme. Convergence history: L_2 norm of the residuals.

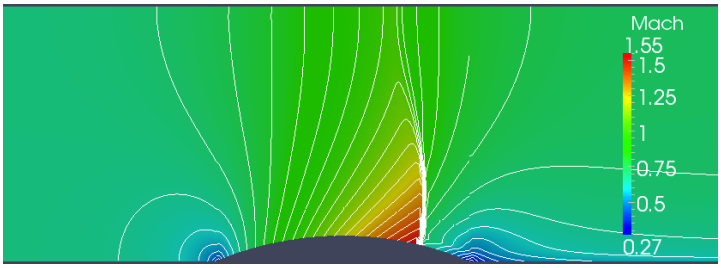


(a)

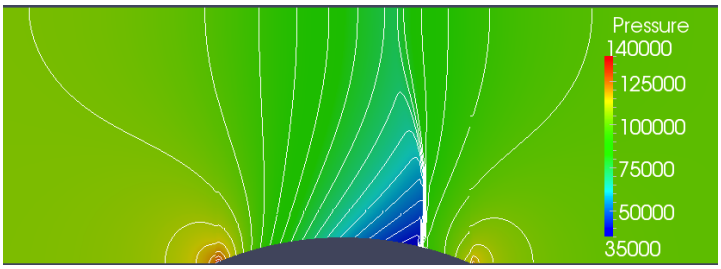


(b)

Figure 7.12: Channel testcase, with our modified Roe scheme. Pressure (a) and Mach contour (b).



(a)



(b)

Figure 7.13: Channel testcase, with our modified Roe scheme. Pressure (a) and Mach contour (b).

7.4 Compression corner testcase

This test case is an example of a supersonic flow over a wedge of angle 15 degree which generates an oblique shock. The Mach number upstream of the shock is fixed to 2.5. This test case has an exact analytical solution, satisfying the Rankine–Hugoniot relations [18], formed by two regions of constant states, separated by the oblique shock. According to the analytical solution the downstream Mach number is equal to 1.87, while the ratio between upstream and downstream pressure is equal to 2.47.
We have used a CFL equal to 20. In fig. 7.14 is shown the convergence history in the L_2 norm of the residuals. In fig. 7.15 are shown the pressure and Mach contours, where we can observe the good agreement with the analytical solution.

7.5 Flat plate testcase

One of the most popular applications of laminar viscous flows is the boundary layer development along a flat plate. The main advantages of this case are its relevance for a number of practical flow problems and the availability of an exact solution, obtained by solving the Blasius equation. The computational domain is composed of:

- An inlet boundary where atmospheric condiction are considered, with Mach number equal to 0.2.
- A downward wall, representative of the flat plate. Its length is fixed to 0.2 meters.
- A backward outlet where the atmospheric pressure is fixed.

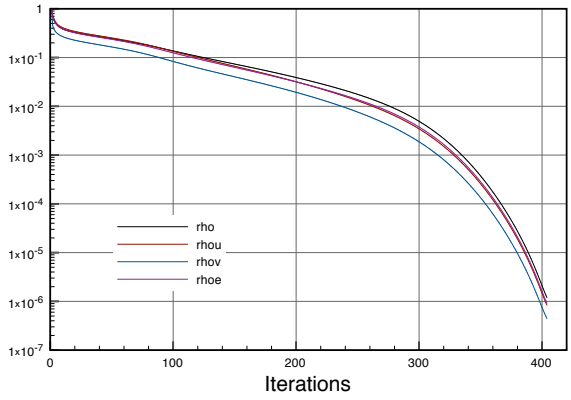
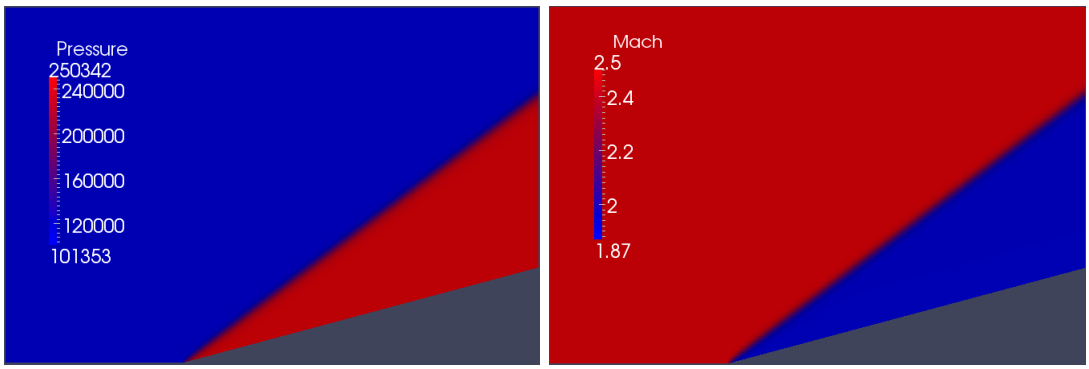
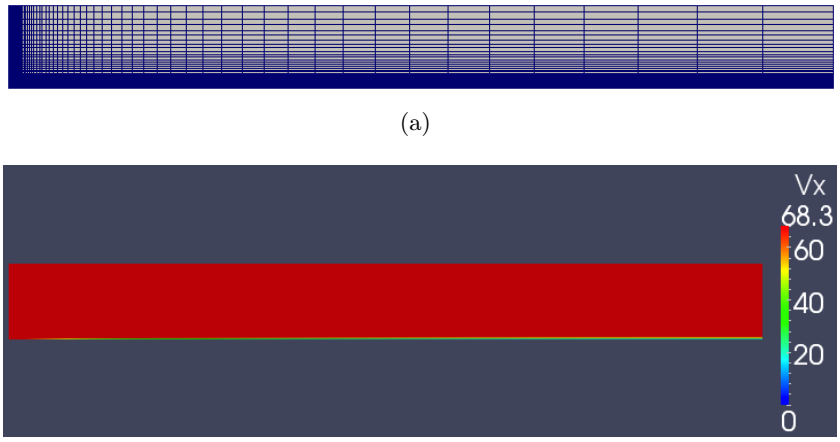


Figure 7.14: Compression corner testcase, with our modified Roe scheme. Convergence history: L_2 norm of the residuals.



(a) (b)

Figure 7.15: Compression corner testcase, with our modified Roe scheme. Pressure (a) and Mach (b) views.



(a)

(b)

Figure 7.16: FlatPlate testcase, with our modified Roe scheme. Mesh (a) and x-direction velocity view (b).

- A upward outlet where the atmospheric pressure is fixed too.

The distance between the upward outlet and the downward wall is established according to the boundary layer thickness. The Reynolds number based on the free stream velocity and the plate length is $8.7 \cdot 10^5$. The boundary layer thickness is computed according to:

$$\delta_{\text{inf}} \cong 5\frac{x}{\sqrt{Re_x}} \tag{7.1}$$

Therefore at $x = 0.2\ m$, that is at the end of the plate, the boundary layer thickness is of the order of 1 mm. We decided than to fix the distance between the wall and the upward outlet equal to 20 times the boundary layer thickness, that is: $0.02\ m$.
An exponential grading in the mesh normal to the plate is used, we fixed its coefficient equal to 1.083317311; this in order to properly capture the boundary layer phenomena.
In fig. 7.16(a) is shown the mesh used for this test case, while in fig. 7.16(b) is shown the velocity component parallel to the flat plate. More important is the fig. 7.17 where the vertical distribution of the orizontal velocity component obtained numerically and analitically (with the Blasius theory) are compared: we can observe a very good matching.

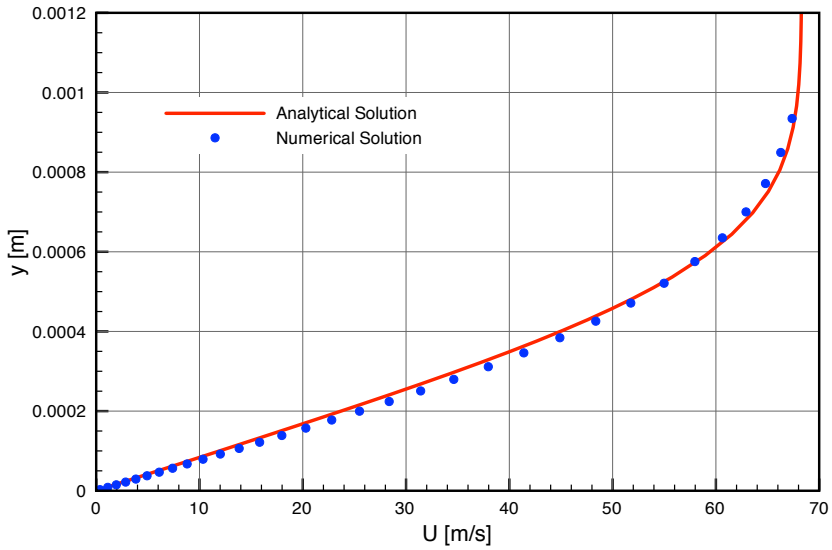


Figure 7.17: FlatPlate testcase, with our modified Roe scheme. Comparison between the Blasius (analytical) solution and the numerical solution.

Part III

Radiative Heat Transfer Modeling and Simulation

Preamble of Part III

Energy transfer by radiation through translucent media with high entalpy can be significant [85]. Media can emit, absorb or scatter radiation, giving rise to a complex phenomenon of heat transfer. The relevance of this phenomenon rises with high temperature. Energy transfer by Radiation has received incresed attention in the past fifty years and many authors have ventured in its study. Some important applications are hot gases in furnaces, engine combustion chamber at high pressure and temperature, rocket propulsion, glass manufacturing, fibrous insulating layers, nuclear explosion, hypersonic shock layer, plasma generators, ablating thermal protection systems, translucent ceramics at high temperature, and heat transfer in porous regions.

Two difficulties make it challenging to study radiation in absorbing, emitting and scattering media [122]. One is that absorption, emission, and scattering can occur at all locations within the medium. A complete solution for energy exchange requires knowing the radiation intensity, temperature, and physical properties throughout the medium. The mathematics describing the radiative field is inherently complex. A second difficulty is that spectral effects are often much more pronounced in gasses than for solid surfaces, and detailed spectrally dependent analysis may be required.

In this third part of the thesis is described our development and the implementation of the Monte Carlo Radiative Transfer Method. This implementation allows to deal with radiative transfer simulations with both 2D or 3D cases, and structured or unstructured grids in parallel computations. This has become possible through the development of algorithms able to perform different tasks efficiently.

The whole is more than the sum of the parts.

Metaphysica
Aristotele

Chapter 8

Radiative Heat Transfer Modeling

8.1 Energy Equation

The energy balance equation for an infinitesimal element reads [85]:

$$\rho C_p \frac{DT}{D\tau} \equiv \rho C_p \Big(\frac{\partial T}{\partial t} + (\mathbf{v} \cdot \nabla) T \Big) = \frac{Dp}{D\tau} - \nabla \cdot (k \nabla T + \mathbf{q}_r) + q_h + \Phi \tag{8.1}$$

where ρ indicates density; C_p, specific heat under constant pressure; T, temperature; τ, time; p, pressure; k, thermal conductivity; $\mathbf{q}_r$, radiant heat flux vector; q_h, chemical heat generation rate; and Φ, viscous heat dissipation rate.
Equation 8.1 expresses the evolution of the temperature along the time. Like all evolution equations it is commonly solved discretizing separatly the space and the time, that is: marching temperature in time. This means: iterativelly computing the temperature along the time. The $\nabla \cdot \mathbf{q}_r$ term in eq. (8.1) signifies the divergence of radiative heat flux, that is the difference between total radiation incident from all solid angles ω_i that is locally absorbed, and the locally emitted radiation. These two quantities (absorbed and emitted radiation) are expressed by the two integral terms over all wavelengths on the right side of the following relation:

$$-\nabla \cdot \mathbf{q}_r = \underbrace{\int_{\lambda=0}^{\infty} a_\lambda(\lambda) \left[\int_{\omega_i=0}^{4\pi} i_\lambda(\lambda, \omega_i) d\omega_i \right] d\lambda}_{absorbed\ radiation} - \underbrace{4\pi \int_{\lambda=0}^{\infty} a_\lambda(\lambda) i_{\lambda b}(\lambda) d\lambda}_{emitted\ radiation} \tag{8.2}$$

where a_λ is the absoption coefficient; λ is the wavelength; ω_i, is a solid angle; $i_{\lambda b}$, is the radiation intensity of a blackbody for a given wavelength; and $i_\lambda(\lambda, \omega_i)$, is the radiation intensity coming to the element which is in function of both the wavelength (λ) and the direction (ω_i). The last term, $i_\lambda(\lambda, \omega_i)$, is the most difficult to determine; in order to compute it the transport equation has been formulated.

8.2 Radiative Transfer Equation (RTE)

Radiative Transfer Equation (RTE) is the equation that describes the radiation intensity, $i_\lambda(\lambda, \omega_i)$, along a path in a fixed direction through an absorbing, emitting and scattering medium.
Lets consider radiation of intensity $i_{lambda}(S)$ along a path S within an absorbing, emitting and scattering medium. As radiation passes through distance dS along S its intensity is decreased by absorption:

$$di_{\lambda,a}(S,\omega) = -a_\lambda(S) i_\lambda(S,\omega) dS \tag{8.3}$$

If the radiation along the path is in local thermodynamic equilibrium, the contribution to the intensity in the S direction by spontaneous emission along dS is given by:

$$di_{\lambda,e}(S,\omega) = a_\lambda(S) i_{\lambda,b}(S) dS \tag{8.4}$$

The attenuation and gain by i_λ in the S direction by scattering is:

$$di_{\lambda,s}(S,\omega) = -\sigma_{s\lambda}(S) i_\lambda(S) + \frac{\sigma_{s\lambda}(S)}{4\pi} \int_{\omega_i=0}^{4\pi} i_\lambda(S,\omega_i)\Phi(\omega,\omega_i) d\omega_i \tag{8.5}$$

Then summing up equations (8.3),(8.4) and (8.5), and deriving over dS we get:

$$\frac{di_\lambda}{dS} = -\underbrace{a_\lambda(S) i_\lambda(S,\omega) dS}_{Loss\ by\ absorption} + \underbrace{a_\lambda(S) i_{\lambda,b}(S) dS}_{Gain\ by\ emission} - \underbrace{\sigma_{s\lambda}(S) i_\lambda(S)}_{Loss\ by\ scattering} + \underbrace{\frac{\sigma_{s\lambda}(S)}{4\pi} \int_{\omega_i=0}^{4\pi} i_\lambda(S,\omega_i)\Phi(\omega,\omega_i) d\omega_i}_{Gain\ by\ scattering} \tag{8.6}$$

The two terms of decrement by absorption and scattering are combined, giving:

$$\frac{di_\lambda}{dS} = -K_\lambda(S) i_\lambda(S,\omega) dS + a_\lambda(S) i_{\lambda,b}(S) dS + \frac{\sigma_{s\lambda}(S)}{4\pi} \int_{\omega_i=0}^{4\pi} i_\lambda(S,\omega_i)\Phi(\omega,\omega_i) d\omega_i \tag{8.7}$$

where $K_\lambda(S) = a_\lambda(S) + \sigma_{s\lambda}(S)$ is the extinction coefficient and in general is a function of position S. We can define the "albedo" function as:

$$\Omega_\lambda = \frac{\sigma_{s\lambda}}{K_\lambda} \tag{8.8}$$

and the optical depth as:

$$k_\lambda(S) = \int_0^S K_\lambda(S)dS \tag{8.9}$$

Equation (8.7) in terms of optical depth and albedo is:

$$\frac{di_\lambda}{dk_\lambda} = -i_\lambda(k_\lambda) + (1-\Omega_\lambda)i_{\lambda b}(k_\lambda) + \frac{\Omega_\lambda}{4\pi}\int_{\omega_i=0}^{4\pi} i_\lambda(k_\lambda,\omega_i)\Phi_\lambda(\omega,\omega_i)d\omega_i \tag{8.10}$$

The final two terms in eq. (8.10) are often combined into the *source function* defined as:

$$I_\lambda(k_\lambda,\omega) \equiv (1-\Omega_\lambda)i_{\lambda b}(k_\lambda) + \frac{\Omega_\lambda}{4\pi}\int_{\omega_i=0}^{4\pi} i_\lambda(k_\lambda,\omega_i)\Phi_\lambda(\omega,\omega_i)d\omega_i \tag{8.11}$$

considering equations (8.10) and (8.11) we get the classical form of the equation of radiative transfer:

$$\frac{di_\lambda}{dk_\lambda} + i_\lambda(k_\lambda) = I_\lambda(K_\lambda,\omega) \tag{8.12}$$

An integrated form of the radiative transfer equation is obtained by using an integrating factor. Multiplying eq. (8.12) by e^{k_λ} gives:

$$\frac{di_\lambda}{dk_\lambda}e^{k_\lambda} + i_\lambda(k_\lambda)e^{k_\lambda} = I_\lambda(K_\lambda,\omega)e^{k_\lambda} \tag{8.13}$$

Integrating over an optical thickness from $k_\lambda = 0$ to $k_\lambda(S)$ and rearranging gives:

$$i_\lambda(k_\lambda,\omega) = i_\lambda(0,\omega)e^{-k_\lambda} + \int_0^{k_\lambda} I_\lambda(k_\lambda,\omega)e^{-(k_\lambda-k_\lambda^*)}dk_\lambda^* \tag{8.14}$$

where k_λ^* is a dummy optical variable of integration along S, and $i_\lambda(0,\omega)$ is the intensity in the direction of S at location where $S = 0$. Eq. (8.14) is the integrated form of the equation of radiative transfer. It is interpreted physically as the intensity at optical depth k_λ, being composed of two terms. The first is the attenuated initial intensity that arrives at k_λ. The second is the intensity at k_λ resulting from emission and incoming scattering in the S direction by all thickness elements along the path from 0 to S, reduced by exponential attenuation between each location of emission and incoming scattering k_λ^* and the location k_λ.

8.3 Solution Procedures

In order to solve the RTE several methods have been developed in the last 50 years [85], [121], [84], [122], [61] and [100]. These methods can be grouped into two types, depending if they are solving the differential or the integral form of the RTE.

Among the differential solution we can cite: the *Milne-Eddington method*, *Diffusion method*, *Taylor series expansion*, and the *Two-flux method*. But the most popular solutions are the integral ones. These can be further classified in:

- **Angular discretization about a volume element**. Like the *Multi-flux method* and the *Discrete transfer metod*.
- **Direct transfer among volume elements**. Like the **Zonal method**, the **Finite Volume, Element or Difference method**.
- **Approximate methods**. Like the *Cold medium and emission approximations*, or the *Mean beam length*.
- **Stochastical solution method**. Like the *Monte Carlo method* or the *Markov chain*.

8.4 Monte Carlo Method

The macroscopic physical processes that can be observed in nature are often quite complicated. However, they derive, in most cases, by the sum of the effects of a large number of processes based on simple nature. Physicists say that in these cases the whole (the overall process) is equal to the sum of the parts (the basic processes). When that is not the case for reasons related primarily to non linear problems, it is said in this case that the whole is not equal to the sum of the parts.
For example, the study of highway traffic can be reduced to studying the behavior of individual cars. Thus simulating the movements of individual cars (basic processes) is possible to deduce the flow rate of road (overall process). Is important to note that at the microscopic level of individual cars, there are no variables that are similar to density and temperature. Conversely, if you look at the macro level we see that for the entire highway one can define variables such as density of cars, and volume of highway occupied by cars, but also temperature: as a measue of the number of small movements that cars make when there is traffic and they are very close to each other and they move slowly and for some few yards at a time. It is said that these variables emerge at the transition from the microscopic to the macroscopic point of view.
For the study of these physical phenomena one can thus use two approaches. In classic one, we study the overall process through the formulation of laws that bind the macroscopic variables. In contrast, in the second approach we study the overall process by simulating the interactions of many basic processes under which it consists. This second approach has the advantage that the basic processes have a simpler physical modeling and are therefore more easily implementable in a code. But it has the disadvantage that, in general, for the emergence of the overall process it is necessary to simulate the interaction of a large number of basic processes. This is not always possible, despite the computational power of computers is progressing rapidly. In this sense scientists have introduced statistical methods to try to overcome this difficulty. The most popular of these is the method of Monte Carlo (MC).
The MC method is a stochastic method for numerical integration. There is no single Monte Carlo method, there are many different approaches instead. The common pattern of all of these approaches tends to have the following steps:

1. Define a domain of possible inputs or problem space.
2. Generate N random "points" x_i in the problem space using a certain specified probability distribution.
3. Perform a deterministic computation of the "score" $f_i = f(x_i)$ for the N "points".
4. Aggregate the results of the individual computations, f_i, into the final result g: $g = g(f_i)\ \forall i$.
5. According to the Central Limit Theorem, for large N g will approach the true value $\hat{g}$.

In this way we avoid to compute all the points (or basic processes) of the problem space (domain of the overall process). This would be computationally too expensive. Instead we compute only a certain number of points, randomly chosen, and relying on the central limit theorem, to ensure the consistency, we can get a good estimation of the value of the global variable that we are looking for.
These concepts are applied in the radiation transfer problem in the following way. The global or whole process is the set of radiative heat transfer phenomena inside the computational domain. The basic process instead, consists in a energy particle emitted from an atom or molecule and absorbed by another atom or molecule. Not being able to simulate, for obvious reasons, the atomic scale, one implements a discretization where the units that emit or absorb the energy particle are no longer infinitesimal size entities such as atoms or molecules, but cells of finite size. These cells can be the same used to discretize the fluid domain (like the finite volume or element cells). Obviously the consistency between finite cells and atoms scales has to worth. For each cell the problem space is the set of energy particles emitted from that cell, therefore each energy particle is a point in the Monte Carlo explanation above. Then, what Monte Carlo method does, is to traces the behavior of a randomly selected finite number of energy particles. Some authors refer to energy particles as rays or beams or virtual photons; hereafter we will use these terms interchangeably.

8.4.1 Simulation of Radiative Heat Transfer

For each cell of the computational domain a heat balance equation is written. For a gas cell we have:

$$q_{r,NET,Gas} = q_{r,IN,Gas} - q_{r,OUT,Gas} \tag{8.15}$$

and for a wall face:

$$q_{r,NET,Wall} = q_{r,IN,Wall} - q_{r,OUT,Wall} \tag{8.16}$$

Where $q_{r,NET,Gas}$ $q_{r,NET,Wall}$ mean, respectivelly, the net radiative heat flow for a gas cell and for a wall face. The radiative energy emitted from a gas cell of volume ΔV is done by:

$$q_{r,OUT,Gas} = 4\sigma K T_g^4 \Delta V \tag{8.17}$$

while the radiative energy emitted from a wall element of a wetted area ΔS is:

$$q_{r,OUT,Wall} = \varepsilon \sigma T_w^4 \Delta S \tag{8.18}$$

The tricky part now is how to compute the absorbed terms of the balance equations. We will discuss how to in solution method (8.4.7). The task of simulating radiative transfer consists of six steps:

1. determine number and energy of the energy particles;
2. simulation of Gas and Wall Emission;
3. ray tracing;
4. simulation of Gas Absorption;
5. simulation of Wall Absorption and Reflection;
6. solution Method.

In the next section we will briefly discuss each of these steps.

8.4.2 Determine number and energy of the energy particles

The first task is the computation of the energy emitted from each cell; then you'll need to assign this energy to a certain number of photons. Here you have two main possibilities.

1. We set the number of energy particles to emit is set constant for all cells. Therefore energy particles of different cells will carry a different amount of energy.
2. The amount of energy that each energy particles can carry is set to constant. Therefore each cell emits a different number of energy particles.

8.4.3 Simulation of Gas and Wall Emission

According to the explanation of the Monte Carlo method given above, in this step the energy particles emitted from each cell have to be sent in random directions **d**s. The direction vector **d** is computed from a set of evenly distributed random numbers: one for each dimension of the physical domain; this according to:

$$\mathbf{d} = \frac{\mathbf{d_r}}{|\mathbf{d_r}|} \tag{8.19}$$

where, for example in 3D:

$$\mathbf{d_r} = \begin{pmatrix} -1 + 2R_1 \\ -1 + 2R_2 \\ -1 + 2R_3 \end{pmatrix}$$

Where the R_i are some evenly distributed random numbers. In order to accomplish this task evenly distributed random numbers have to be generated. This has to be done carefully. In fact, in order to properly use the Monte Carlo method it is necessary that the random number generator is powerful enough, that is: the numbers generated have to be able to pass sophisticated tests of randomness. This is a key point for the success of the method. In Paragraph 8.5 it will be explained how to generate these random numbers.

8.4.4 Ray Tracing

Once the energy particle is calculated for each direction, one must trace his path within the computational domain until one of the following three things happen:

- the energy particle is absorbed into a gas cell;
- the energy particle is absorbed on a wall face;
- the energy particle disappears in another boundary.

In order to accomplish this task we have developed and implemented a particle tracking algorithm, more widely explained in Chapter (9).

8.4.5 Simulation of Gas Absorption

Let's consider the equation for absorption (8.3), and let's integrate it for the case of radiant energy with intensity I_0, coming within a solid angle $d\Omega$, that enters into a gas volume of thickness S and cross-sectional area dF. With these assumptions we get:

$$Id\Omega dF = I_0 d\Omega dF e^{-a_\lambda S} \tag{8.20}$$

and simplifying we obtain the BEER's Law that expresses the attenuation of radiant energy inside a volume of thickness S as:

$$I = I_0 e^{-a_\lambda S} \tag{8.21}$$

Here, the physical units of a_λ and S are inverse meters and meters, respectivelly. Their product $a_\lambda S$ becomes dimensionless, and is named absorptive distance or optical length.
Equation (8.21) treats the radiative energy as a continuous as it has to be in a macroscopic point of view (wath we called whole process). But with the Monte Carlo method we are no longer modeling the macroscopic phenomena because of their laws, but as a superposition of the basic microscopic processes. This procedure is called distributed approach and no longer needs the use of a continuous variable as the intensity of the radiation. But what the distributed approach has to ensure is that with an adequate number of simulated energy particles we get the behavior described by macroscopic laws. In the case of absorption we have to ensure that from the laws used to describe it, the individual energy particles naturally follow the Beer's law.

It can be shown that to achieve this purpose it is sufficient that the individual energy particles yield obedience to the two following postulates:

1. all radiative energy of each particle will eventually be absorbed by gas atoms or molecules at a certain location x;
2. the energy possessed by each particle remains unchanged during its flight.

This is precisely the same concept as the transfer of radiative energy by "real" photon. The energy particle absorption happens when during its flight the photon optical length reach a threshold value, $a_\lambda S$, of the optical length. This threshold value is given by:

$$a_\lambda S = -ln(1 - R_s) \tag{8.22}$$

Where R_s is a distributed random number; so, as for the emission step here also we need to use the random number generator. It can be shown that equation (8.22) is the distributed version of Beer' law (8.21). Therefore the absorption criteria is:

$$\int_0^S a_\lambda ds \geq -ln(1 - R_s) \tag{8.23}$$

This is one of the criteria applied by the ray tracing algorithm to stop the tracking of the particle, see Chapter (9).

8.4.6 Simulation of Wall Absorption and Reflection

In the continuous approach, when a beam impacts against a wall it happens that some of its energy is absorbed by the wall and another part remains in the beam that is reflected. Unlike in the distributed approach of Monte Carlo the impacting energy particle can be either completely absorbed or completely reflected. All the energy possesed by the energy particle goes respectively to the wall or to the reflected energy particle. The criterion that discriminates the two cases is the following:

$$R_r \begin{cases} \leq \varepsilon & \text{then the energy particle is absorbed;} \\ \geq \varepsilon & \text{then the energy particle is reflected.} \end{cases} \tag{8.24}$$

Where ε is the emissivity of the wall. If reflection happens, the energy particle is reflected with the distributed counterpart of the Lambert's cosine law. That is with the equations:

$$\theta = 2\pi R_\theta \tag{8.25}$$
$$\eta = acos(sqrt1 - R_\eta) \tag{8.26}$$

where R_θ and R_η are again evenly distributed random numbers; while θ and η are spherical cordinate. In case of perfectly reflecting wall, like a mirror, we can apply the specularly reflection rule.

8.4.7 Solution Method

Each time a particle is absorbed by a wall face or a gas cell we have to stop tracking it and store the informations related to: energy possessed, cell or face that emitted it and cell or face that has assorbed it. Once for each cell or face, all particles have been traced we can compute the heat flux entering in each element. Two methods are available to do that: the energy method and the radiative energy absorption distribution (READ) method. We will briefly discuss both of them.

Energy method

In this method the energy e_0 possesed by every energy particle is the same for all cells. Therefore, what changes between different cells is only the number N_a of energy particles emitted. Then we have:

$$q_{r,IN} = N_a e_0 \tag{8.27}$$

From this equation it is clear that N_a is a function of the leaving radiative heat flux and then it is in function of the temperature, see equations (8.17), (8.18). This means that the Monte Carlo procedure has to be applied for each iteration in the temperature convergence loop. Obviously this is not the best approach from a computational time consuming point of view.

READ method

In order to avoid that the Monte Carlo procedure is applied for each iteration, some authors developed the READ method. In this work a particular version of the READ method has been developed in which the incoming radiative heat flux is computed as:

$$q_{r,IN,i,b} = \sum_{gas} R_d(a,b) \cdot q_{r,OUT,Gas,a} + \sum_{wall} R_d(a,b) \cdot q_{r,OUT,Wall,a} \tag{8.28}$$

$for\ i = gas,\ wall$. Here, $R_d(a,b)$ is the fraction of radiative energy emitted from all gas and wall elements (indicated with a) excluding the element under consideration, that is absorbed by the element (indicated with b).
This method relies on the fact that the magnitude of the statistical pattern $R_d(a,b)$ depends only on the system geometry and the distribution of radiative physical properties and not on the temperature. It follows that we can perform the computation of $R_d(a,b)$ by means of Monte Carlo method just once at pre temperature iteration step.

8.5 Random Number Generation

8.5.1 Pseudo-random numbers

In essence, there is no such a thing as a single random number. Rather, we speak of a sequence of random numbers that follow a specified theoretical or empirical distribution. There are two main approaches to generate random numbers. In the first approach, a physical phenomenon is used as a source of randomness from where random numbers can be generated. Random numbers generated in this way are called true random numbers.
An alternative approach to generating random numbers, which is the most popular approach, is to use a mathematical algorithm. Efficient algorithms have been developed that can be easily implemented in a computer program to generate a string of random numbers.
These algorithms produce numbers in a deterministic fashion. That is, given a starting value, called the seed, the same sequence of random numbers can be produced each time as long as the seed remains the same. Despite the deterministic way in which random numbers are created, these numbers appear to be truly random since they pass a number of statistical tests designed to test various properties of random numbers. In view of this, these random numbers are referred to as pseudo-random numbers. An advantage of generating pseudo random numbers in a deterministic fashion is that they are reproducible, since the same sequence of random numbers is produced each time we run a pseudo-random generator given that we use the same seed. This is helpful when debugging a simulation program, as we typically want to reproduce the same sequence of events in order to verify the accuracy of the simulation.
Pseudo-random numbers and in general random numbers are typically generated on demand. That is, each time a random number is required, the appropriate generator is called which then returns a random number. Consequently, there is no need to generate a large set of random numbers in advance and store them in an array for future use as in the case of true random numbers.
In general, an acceptable method for generating random numbers must yield sequences of numbers or bits that are:

- uniformly distributed,
- statistically independent,
- reproducible, and
- non-repeating for any desired length, where this length is called the "period" of the method.

The Congruential Method

This is a very popular method and most of the available computer codes for the generation of random numbers use some variation of this method. The advantage of this congruential method is that it is very simple, fast, and it produces pseudo-random numbers that are statistically acceptable for computer simulation.
The congruential method uses the following recursive relationship to generate random num-

bers.

$$x_{i+1} = ax_i + c(mod\ m) \tag{8.29}$$

where x_i, a, c and m are all non-negative numbers. Given that the previous random number was x_i, the next random number x_{i+1} can be generated.
The number of successively generated pseudo-random numbers after which the sequence starts repeating itself is called the *period*.

General Congruential Methods

The congruential method described above can be thought of as a special case of the following generator:

$$x_{i+1} = f(x_i, x_{i-1}, ...)(mod\ m) \tag{8.30}$$

where f(.) is a function of previously generated pseudo-random numbers. A special case of the above general congruential method is the quadratic congruential generator. This has the form:

$$x_{i+1} = a_1 x_i^2 + a_2 x_{i-1} + c(mod\ m). \tag{8.31}$$

Composite generators

We can develop composite generators by combining two separate generators (usually congruential generators). By combining separate generators, one hopes to achieve better statistical behavior than either individual generator.

8.5.2 The Mersenne Twister

Below we present a description of the Mersenne-Twister algorithm following the paper ([78]).
The *Mersenne twister* (MT) is an important pseudo-random number generator with superior performance. Its maximum period is $2^{19937} - 1$, which is much higher than many other pseudo-random number generators, and its output has very good statistical properties. The MT generates a sequence of bits, which is as large as the period of the generator after which it begins to repeat itself. This bit sequence is typically grouped into 32-bit blocks (i.e., blocks equal to the computer word). The blocks are considered to be random.
The following is the main recurrence relation for the generation of random sequence of bits:

$$x_{k+n} = x_{k+m} \oplus (x_k^u \mid x_{k+1}^l)A \tag{8.32}$$

We assume that each block, represented by x, has a size of w bits. The meaning of the parameters used in the above equation is as follows:

- x_k^u: the upper $w - r$ bits of x_k, where: $0 \leq r \leq w$.
- x_{k+1}^l: the lower r bits of x_{k+1}.
- $\oplus$: exclusive OR.

- |: it indicates the concatenation (i.e., joining) of two bit strings.
- n: degree of recurrence relation.
- m: integer in the range $0 \leq m \leq n$.
- A: a constant w x w matrix defined as below so that the multiplication operation in the above recurrence can be performed extremely fast.

$$\begin{pmatrix} 0 & 1 & \dots & 0 & 0 & 0 \\ 0 & 0 & \dots & 0 & 0 & 0 \\ \dots & \dots & \dots & \dots & \dots & \dots \\ 0 & 0 & \dots & 0 & \dots & 0 \\ 0 & 0 & \dots & 0 & 0 & 1 \\ a_{w-1} & a_{w-2} & \dots & \dots & \dots & a_0 \end{pmatrix}$$

The recurrence relation is initialized by providing seeds for the first d blocks, i.e., $x_0, x_1, ..., x_{n-1}$. The multiplication operation xA can be done very fast as follows:

$$xA = \begin{cases} shiftright(x) & \text{if } x_0 = 0, \\ shiftright(x) \oplus a & \text{if } x_0 = 1. \end{cases} \tag{8.33}$$

where $a = \{a_{w-1}, a_{w-2}, \dots, a_0\}$ and $x = \{x_{w-1}, x_{w-2}, \dots, x_0\}$. At the last state of the algorithm, in order to increase the statistical properties (in particular the equidistribution) of generator's output, each generated block is multiplied from the right with a special w x w invertible tempering matrix T. This multiplication is performed in a similar manner as the multiplication with matrix A above and it involves only bitwise operations, as follows.

$$y = x \oplus (x >> u) \tag{8.34}$$

$$y = y \oplus ((y << s)b) \tag{8.35}$$

$$y = y \oplus ((y << t)c) \tag{8.36}$$

$$z = y \oplus (y >> l) \tag{8.37}$$

where:

- b and c are block size binary bitmasks (vector parameters).
- l, s, t and u are pre-determined integer constants.
- $(x >> u)$ indicates a shiftright operation by u times for variable x.
- $(x << u)$ indicates a shiftleft operation by u times for variable x.

In order to have a period of $2^{19937} - 1$ and good statistical property, the Mersenne-Twister's authors suggest the following values for the constant.

- $(w, n, m, r) = (32, 624, 397, 31)$,
- $a = 0x9908BoDF$,

- $u = 11$,
- $s = 7$,
- $b = 0x9D2C5680$,
- $t = 15$,
- $c = 0xEFC60000$,
- $l = 18$.

With these values the method is identified as *MT19937*.

8.5.3 Implementation

Between different possible Mersenne-Twister Methods, we decide to implement the MT19937, since it is considered the most performing one, and the most suitable for applications in the Monte Carlo Method.
The implementation has been made in the C++ language, see Chapter (2). The criteria which the code is written with, respond to the need of computational velocity.
As explained above the MT algorithm works following these steps.

Inizialization First of all it requires a seed with which according to a certain strategy it generates a sequence of $n(= 624)$ numbers, in bit format. The seed could be an arbitrary number or an arbitrary vector. The n numbers, generated in this way, are stored in a state vector s.

State Generation Given the state vector s a sequence of n pseudo-random numbers are generated appling Eq. (8.32)–(8.37). Once the new n random numbers are generated they constitute the updated state vector s.

Utilise When the user needs a random number, the algorithm gives him an element of the state vector. Once all the n element of the state vector have been gived to an user, the algorithm go back to the State Generation step and generates other n random numbers.

We implement two polymorphic inizialization function called seed. One is seeded with a number, instead the other is seeded with a vector. The constructor of the class can therefore be initialized with a number, a vector or nothing (in this case it uses a default seed number). The state generator is split in the following three functions:

- a function called *recurrence* that implement: $(x_k^u \mid x_{k+1}^l)A$.
- a function called *NewState* that implement Eq. (8.32) using *recurrence*.
- a function called *rand* that make the tempering and decides if a new state generation is needed or not.

The utilise step is accomplished with different functions according to the need of the user.
If in a C++ program one has only a short function, than in order to faster the code it is a nice idea to put all this function inline in the header file. This is convenient only if the functions are short otherwise we get a big code (that is slower in fact). In the MT algorithm the inizialization

and part of the state generation steps happen only once each 624 demand of a random number from the user. Therefore we put these in the implementation file, while all the other functions are made inline.

Simplicity is prerequisite for reliability.

E. Dijkstra

Chapter 9

Particle Tracking

9.1 General Introduction

Monte Carlo Radiative Transfer is not the only frame that needs a particle tracking tool [65]. Indeed, in many applications of computational physics, particles entities are involved. These particles can represent different physical entities, like

- bubbles
- beames
- small solid particles
- virtual entities used to define the flow field

All these entities are moving in the fluid domain in accordance with their laws. Whichever the laws that moves the particle is, the CFD code needs, for its computations, to know the cells that a particle has crossed. In other words: the code needs to locate the particles. Several alternative techniques have been proposed in the literature to efficiently locate particles within unstructured grids.
Among others we can mention the use of:

- A Cartesian background grid.
- Tree structures.
- The successive neighbour searches.

In the cartesian background grid the idea is to superimpose the irregular foreground grid on a regular background grid. The elements of the foreground mesh that cover each cell of the cartesian mesh are stored in a linked list. Given a new particle position x_p, the cell of the cartesian background grid is obtaind from:

$$i_c = INT\left(\frac{(x_p - x_{min})}{(x_{max} - x_{min})}N_x\right) + 1$$
$$j_c = INT\left(\frac{(y_p - y_{min})}{(y_{max} - y_{min})}N_y\right) + 1$$

The main shortcoming of this approach is the inefficiency and inaccuracy that arises when meshes with large variations in element size are employed.
In order to circumvent the problem encountered by the previous method, the tree Structure Method attempts to use a hierarchy of cartesian meshes. The shortcoming now is the additional complexity in coding.
The successive Neighbor Searches Method [75], [119], [123], instead have the following facilities.

- The idea is to exploit as a judicious guess the host element before the particle was moved.
- Then, should the particle no longer be found in that element, the immediate neighbour most likely to contain the particle is searched next.
- This procedure is repeated until the new host elemet is found.
- The algorithm is relatively easy to implement and fast.
- The number of neighbour-element searches required is typically small.
- This method has no problems related to mesh with large variations in element size.

The idea commonly used to search the neighbour element that hosts the particle, is based on the shape functions. A possible algorithm that implements this idea follows these steps:

1. Initial conditions: start point, $\vec{P}$, direction, $\vec{d}$, spatial increment, h.
2. Determine target point T with: $\vec{T} = \vec{P} + h\vec{d}$.
3. Transform the start cell position vector into unit space coordinate.
4. Determine if the target point lies in the same cell, if yes, double the spatial increment, h, and restart with step 2, if not proceed with step 5.
5. Determine the cell face through which the particle will leave the current cell using the transformation in unit space.
6. Given the exit face find the next cell.
7. Use the same method to determine if the target point lies within the adjacent cell, if not halve h and restart at step 2.
8. Now the particle has travelled from the initial cell to a immediate neighbour

However the neighbour search methods based on shape functios have some shortcomings:

- first of all we note that in order to find the host cell, one should go from the physical space to the unit space, make all the tasks needed to find the host cell and come back to the physical plane;
- many different variants of this approach do exist for 2D case, while few robust suggestions have been proposed for 3D case;

- there is the lack of a general implementation valid for both 2D and 3D cases, and for whatever kind of cells are employed;
- therefore in order to deal with every possible mesh, both 2D and 3D, one must deal with a program that implements methods for all possible cases, thus making things more complicated.

On the contrary one would like a general method which is able to deal with:

- 2D case;
- 3D case, without loosing simplicity than if 2D;
- all types of cells, treating them all equally.

A method that meets all these demands is the proposed one based on an adaptation of the method of *R. Chorda, et.al.*, see [29]. This method employs only vector operations and vector properties in order to accomplish its tasks. In the next section we will discuss how this algorithm works

9.2 Particle Tracking Algorithm

In figure (9.1) it is shown a typical case which apply the particle tracking. Here we have the point $x(t)$ where the particle is at time t, and the point $x(t + \Delta t)$ where the particle is at time $t + \Delta t$. The goal of the particle tracking algorithm is to find all and only the cells that are crossed by a straight line connecting point $x(t)$ and $x(t + \Delta t)$.

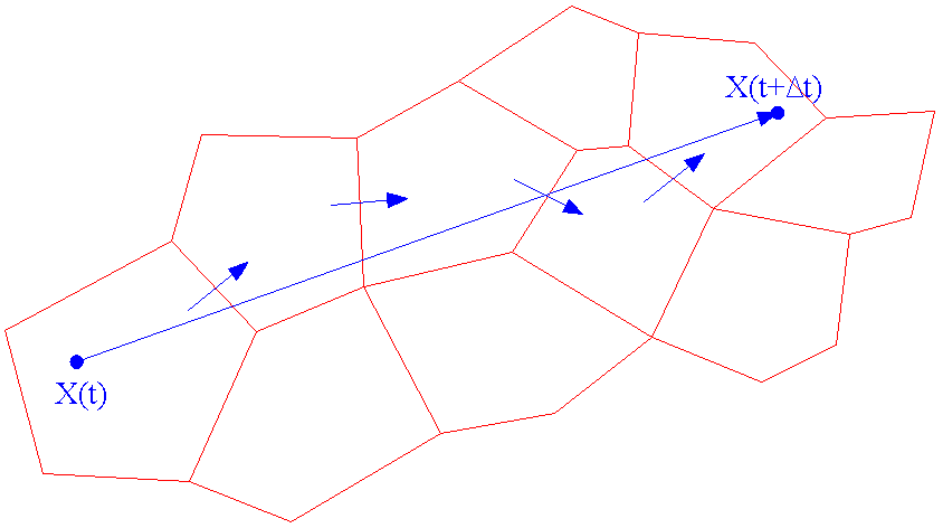


Figure 9.1: Particle Tracking Goal

Now we will describe the algorithm for the 2D and for the 3D case.

9.2.1 2D Tracking

Let's suppose that we know a cell that is crossed by the particle, we call this cell actual cell or current cell. No matter if this is the cell from which the particle starts or ends its flight, or it is a cell in between these two. The proposed algorithm needs the following tools:

- an algorithm that sorts counterclockwise the nodes of all cell faces: sort algorithm;
- an algorithm that checks if a cell face is intersected by the particle trajectory: T2L test algorithm;
- an algorithm that checks if a particle is inside a cell: P2L test algorithm.

Below we explain how we sort the nodes and what P2L and T2L tests mean. Afterward, with an example, we explain how the 2D particle tracking works.

2D sort

In order to apply the successive T2L and P2L tests we need to have the nodes of each cell faces sorted counterclockwise as is depicted in Fig. (9.2).

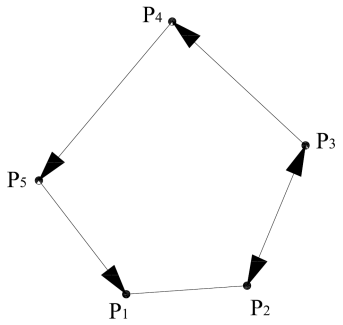


Figure 9.2: Counterclockwise face nodes order

In order to accomplish the sorting task we just operate as is depicted in figure (9.3). Let's call the baricentre point of the cell as G. If the 2 nodes, A and B, of a face are counterclokwise sorted in the sequence "A *then* B", then the cross product $\overrightarrow{GA} \times \overrightarrow{GB}$ has to be positive. If not, then the nodes have to be sorted in the sequence "B *then* A".

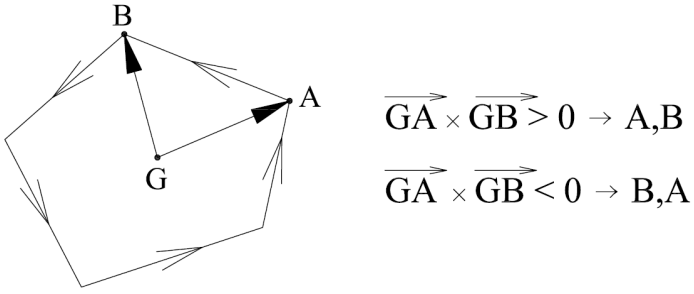


Figure 9.3: sorting face nodes in 2D

T2L test

With the T2Ltest we are able to detect the faces that are intersected by the particle trajectory. How this test works is illustrated in Fig. (9.4). The figure shows how the z component of the cross product of the vector $\overrightarrow{X(t)P_i}$ and $\overrightarrow{X(t)X(t+\Delta t)}$ can be used to detect the face tajectory intersection. The expression of the z component of vertex i is:

$$L_i = (X_i - X(t))(y(t+\Delta t) - y(t)) - (X(t+\Delta t) - X(t))(Y_i - Y(t)) \tag{9.1}$$

Fig. (9.4) shows that $L > 0$ when the particle trajectory lies to the left of a given vertex (e.g., vertex $i+1$), and $L < 0$ otherwise (e.g., vertex i). For this reason, the computation of the value of L_i will be termed as *trajectory to the left* (T2L) test. The figure clearly reveals that if two consecutive vertices have opposite signs of T2L, then the particle trajectory crosses the face connecting such vertices.

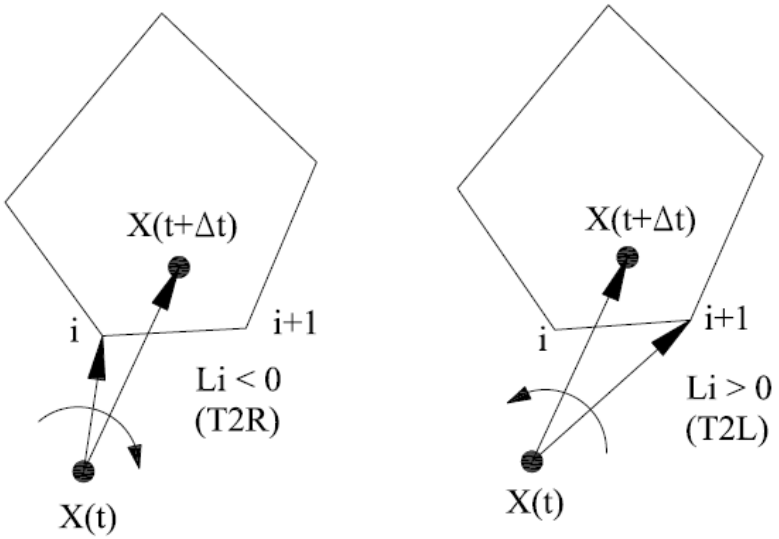


Figure 9.4: T2L test

P2L test

The *particle to the left* (P2L) test aims to find out whether a particle lies within a given cell: move along the two cell faces intersected by the particle trajectory and check if the particle lies to the left of the faces. If this is the case, the particle is within the cell. The P2L condition can be checked by looking at the z component of the cross product between the face vector $\overrightarrow{P_iP_{i+1}}$ and the particle vector $\overrightarrow{P_iX(t+\Delta t)}$, where $X(t+\Delta t)$ is the particle position to be located (if it is inside or outside the current cell).

$$\Omega_i = (X_{i+1} - X_i)(Y(t+\delta t) - y_i) - (X(t+\Delta t) - X_i)(Y_{i+1} - Y_i) \tag{9.2}$$

- $\Omega_i > 0$ indicates that the point P_i is on the left-hand side of the cell face.
- $\Omega_i < 0$ indicates that the point P_i is on the right-hand side of the cell face.
- $\Omega_i > 0$ indicates that the point P_i is on the cell face.

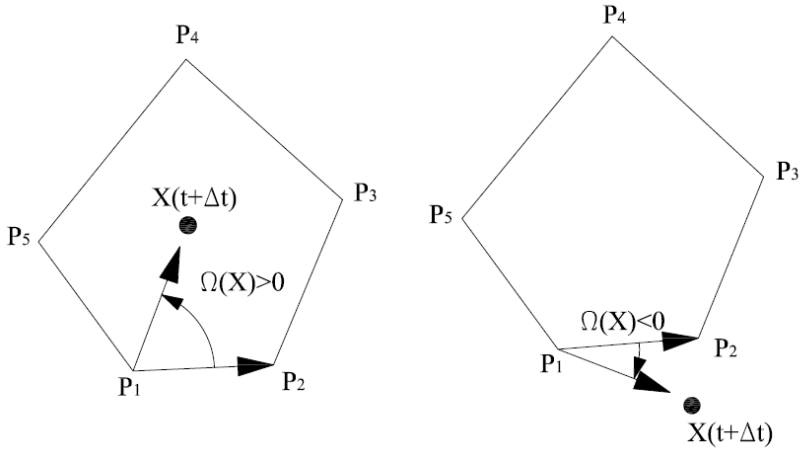


Figure 9.5: P2L test

How it works

The 2D particle tracking algorithm has the following steps:

1. Sort the nodes of each face of the current cell, see (9.2.1).
2. Check if the particle trajectory $\overrightarrow{X(t)X(t+\Delta t)}$ crosses one face of the current cell. This is done by applying the T2L test, see (9.2.1), to the two face vertices and comparing the sign of the T2L test. In the cell I of fig. (9.6), the crossing faces are BC and FA.
3. If it does, check the P2L test, see (9.2.1), on that face. If the particle lies to the right of the face ($P2L < 0$), then we have found the appropriate (i.e., exit) crosssing face (this is the case of cell I, face BC in fig. (9.6)). Exit the loop and move to the neighbouring cell that shares that face.
4. Move to the next face and go to step 2.
5. If the loop over all the cell faces is finished without fulfilling step 3 (i.e., $P2L > 0$ for all the crossing faces, such as faces IJ and CB of cell II of fig. (9.6)), then the particle lies within the current cell.

9.2.2 3D Tracking

Like in the 2D case also in 3D we need three tools:

- an algorithm that sorts counterclockwise the nodes of all cell faces (looking at the face from outside the cell to which it belongs): sort algorithm;

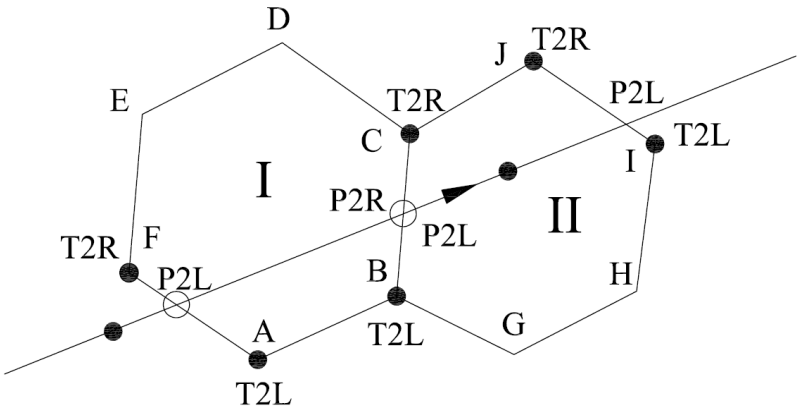


Figure 9.6: Example of the 2D particle tracking algorithm

- an algorithm that checks if a cell face is intersected by the particle trajectory, and, in case, if it is a leaving or entering face: T2I test algorithm;
- an algorithm that checks if a particle is inside a cell: P2I test algorithm.

Below we explain how we sort the nodes and what P2I and T2I tests mean.

3D sort

Given two nodes of a 3D face we can find which one came before in a counterclockwise order with the test depicted in fig. (9.7). In this figure G_c is the barycentre point of the cell; G_f is the barycentre point of the face; A and B are two nodes to be sorted. Let's define the vector variable $L(A, B)$ as:

$$L(A, B) = (\overrightarrow{G_f A} \times \overrightarrow{G_f B}) \cdot \overrightarrow{GcGf} \tag{9.3}$$

One can easily see that: if $L(A, B) > 0$ then B follows A; if $L(A, B) < 0$ then A follows B.

This test allows us to order two nodes at a time. Things get complicated when you have to sort all the nodes of a face together. For example, in fig. (9.7), nodes are originally stored as indicated by the letters that identify them. One may find different algorithms of order $O(N)$,that make their task based on the sign of $L(A, B)$. In our opinion one of the most effective strategies is to use a merge-sort algorithm in which the condition of comparison is based on the sign of $L(A, B)$. This solution is very effective and has an order of $O(nlogN)$.

T2I test

The *trajectory towards the inside* (T2I) test, tells us if a cell face is intersected by the particle trajectory, and, in case, if this face is a leaving or entering face. Like in the 2D case this task is

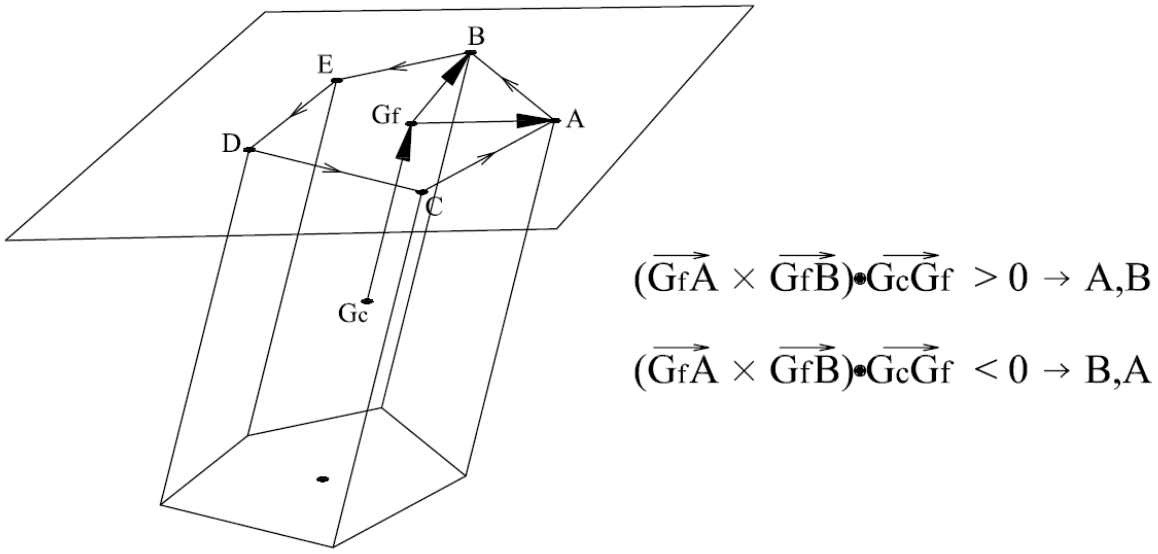


Figure 9.7: sorting face nodes in 3D

accomplished without the need for the expensive computation of face-trajectory intersections; instead, a set of simple algebraic expressions is evaluated. The first step for the detection of trajectory-face intersection is to check the relative orientation of a face segment and the trajectory. This task is shown in Fig. (9.8) for the face segment P_1P_2 and the bottom cell face. The key element for the relative orientation of P_1P_2 and the particle trajectory is the plane that contains the point P_1, P_2 and $X(t + \Delta t)$. Three different situations can be found:

1. Trajectory towards the inside of the face $(\overrightarrow{X(t)X^a(t + \Delta t)})$.
2. Trajectory passing through the face border $(\overrightarrow{X(t)X^b(t + \Delta t)})$.
3. Trajectory towards the outside of the face $(\overrightarrow{X(t)X^c(t + \Delta t)})$.

It is clear that, in a situation like the one portrayed in Fig. (9.8), a particle trajectory crosses a given cell if, and only if, the condition (a) is verified for all the face segments. Thus we need a mathematical expression that classifies the trajectory as type (a), (b) or (c) with respect to each face segment P_iP_{i+1}. For that purpose, we can define the following vectors:

$$\mathbf{a} = \overrightarrow{X(t)P_1} \tag{9.4}$$

$$\mathbf{b} = \overrightarrow{X(t)P_2} \tag{9.5}$$

$$\mathbf{n} = \mathbf{a} \times \mathbf{b} \tag{9.6}$$

$$\mathbf{t} = \overrightarrow{X(t)X(t + \Delta t)} \tag{9.7}$$

Here, $\mathbf{n}$ is the normal vector to the plane $P_1P_2X(t)$. This vector, due to the ordering of the face vertices, always points towards the outside of the face. Hence, the three trajectory types can be identified by projecting the trajectory vector $\mathbf{t}$ on the normal vector $\mathbf{n}$:

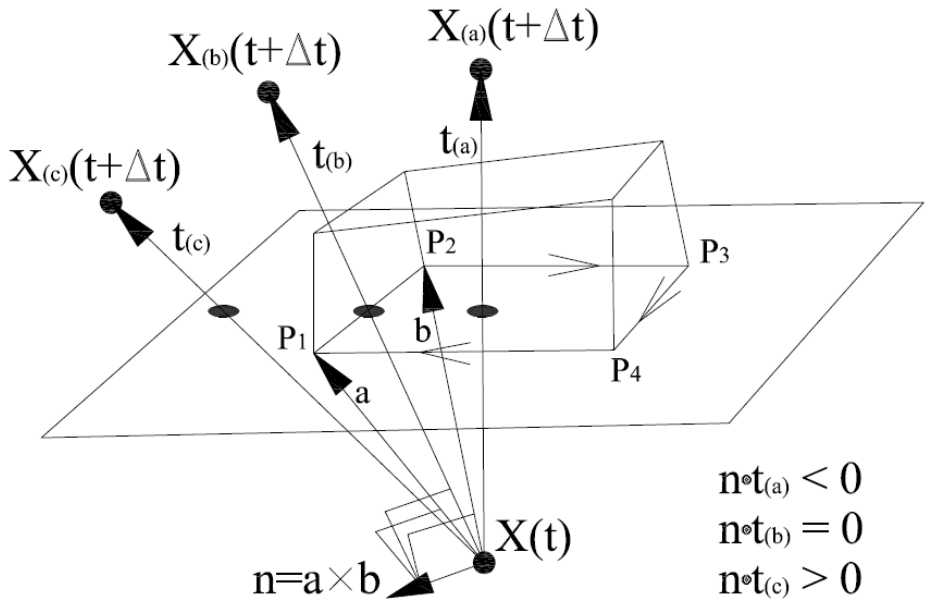


Figure 9.8: Definition of vectors for the detection of 3D trajectory-face intersection (bottom cell face)

1. $\mathbf{n} \cdot \mathbf{t} < 0 \rightarrow$ trajectory towards inside of face.
2. $\mathbf{n} \cdot \mathbf{t} = 0 \rightarrow$ trajectory through face border.
3. $\mathbf{n} \cdot \mathbf{t} > 0 \rightarrow$ trajectory towards outside of face.

A similar analysis can be carried out for the *top* face of the same cell. This analysis is presented in Fig. (9.9). Due to the face-vertices ordering, now the normal vector $\mathbf{n}$ points towards the inside of the face. As a result, a trajectory-face intersection is found if $\mathbf{n} \cdot \mathbf{t} > 0$ for all the face segments $P_i P_{i+1}$.

The method indicated above can be summarized in the following two points:

1. The particle trajectory crosses a given face if and only if:

$$\forall i \; Sign[(\overrightarrow{X(t)P_i} \times \overrightarrow{X(t)P_{i+1}}) \cdot \vec{t}] = const \tag{9.8}$$

2. The sign of $\mathbf{n} \cdot \mathbf{t}$ provides additional information. As the vertices are counterclokwise ordered for each face of a cell, the value of the sign tells us whether the particle leaves or enters the cell through that face, see Figs. (9.9) and (9.8).:

$$\mathbf{n} \times \mathbf{t} > 0 \rightarrow \text{particle leaving cell}$$
$$\mathbf{n} \times \mathbf{t} < 0 \rightarrow \text{particle entering cell}$$

After checking the face-trajectory intersection, one must verify if the final position of the particle is towards the inside the cell. This can be easily done with the P2I test.

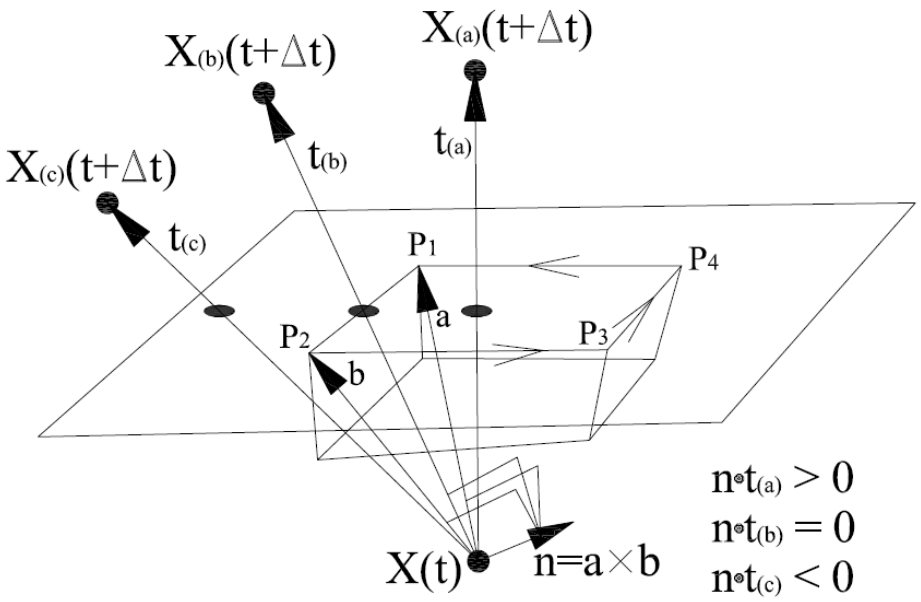


Figure 9.9: Definition of vectors for the detection of 3D trajectory-face intersection (top cell face)

P2I test

The *particle towards the inside* (P2I) test aims to find out whether a particle lies within a given cell: move along the two cell faces intersected by the particle trajectory and check if the particle lies towards the inside of the cell. Assuming always that the face nodes are counterclockwise ordered we define for each face node i the P2I variable, see Fig. (9.10):

$$P2I = Sign[(\overrightarrow{P_{i+2}P_{i+1}} \times \overrightarrow{P_{i+1}P_i}) \cdot \overrightarrow{P_{i+1}X(t+\Delta t)}] \tag{9.9}$$

If, for all face nodes i, P2I is positive, then the particle trajectory is towards the inside respect to that face. Finally if both the intersected cell faces meet the P2I condition, then the particle is inside the cell.

9.3 Ray Tracing

The problem of tracking the particle trajectory within the computational domain can occur in two modes. Depending on whether we know or we do not know in advance the position of the endpoint.

1. The first case is when the end point could be calculated based on the state of the fluid at the point where the particle was at the previous time step. This is the case of a solid particle that moves according to the resultant of forces acting on it at a given instant of time. The tracing algorithm will then find which is the cell that hosts the endpoint of the particle given the coordinates.

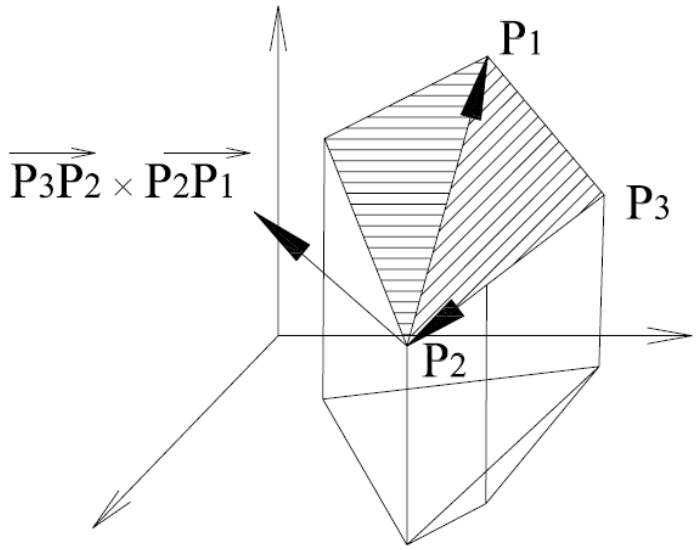


Figure 9.10: Schematic of a 3D cell, showing counterclockwise order for the face nodes.

2. The second case is when the endpoint is derived from the value of a function that integrates gradually along the route. This is the case of an energy particle (in the Monte Carlo Method for Radiative Heat Transfer) that travels in one direction and will stop (it will be absorbed) when its optical length (integral function of the path) reaches a threshold value. Starting from the cell that currently hosts the particle, the tracking algorithm has to find the immediately neighbouring cell where the particle is going.

For the ray tracing we are interested only in the second case, however we decided to develop a program that was general enough to be able to deal with any type of particle; not only energy particle, but also solid particle, bubble, etc. Therefore we implemented both possible cases of particle tracking. This has been done adapting the above particle tracking algorithm in order to have an optimazed implementation for both cases.
Another important implementation detail is that we can ask the program to start the tracking from both the centroid of a cell or from any point inside a cell. In ray tracing the particle will start from the center of a cell when it is emitted by this cell. Instead, it starts from a specific point when it is reflected or comes from a different cell from that which issued it.
The parallelization of the algorithm has been implemented using the double level approach. In this approach we have two programs: the slave and the master. The slave works in serial, while the master performs the parallel tasks. In our implementation the slave performs in serial all the particle tracking algorithm, while the master performs the physical implementation task and the inter-process communications. We found that this approach was the most general, and the most computationally efficient: since it is carrying less communications. When the slave program during the tracing of the path of a particle, reaches the boundary of two processors subdomains, it stops computing that particle and tells what has found to the master program. The master collects this information and once all particles have been traced within the different subdomains processor, it performs a communication for exchanging data between processors.
Moreover as we said the master program performs also the physical computations by the RayTracing function. In the radiation transfer the energy particles are traced by the slave

cell by cell, and each time the slave tells to the master which is the new hosting cell or the face that the particle trajectory crosses. With these informations the master checks the following possibilities:

1. The particle is absorbed by the new hosting cell;
2. The particle has impacted to the wall, and, in that case, if it is reflected or absorbed;
3. The particle has crossed a boundary (different from wall) and then it disappears;
4. The particle has crossed a partition boundary, and then it will have to be sent to the processor that computes the new hosting cell;
5. It continues to flight within the same processor subdomain.

The master function that computes the physical part and acts as interface with the slave has been called *Ray Tracing*. In Fig. (9.11) is depicted the flow diagram of this program.

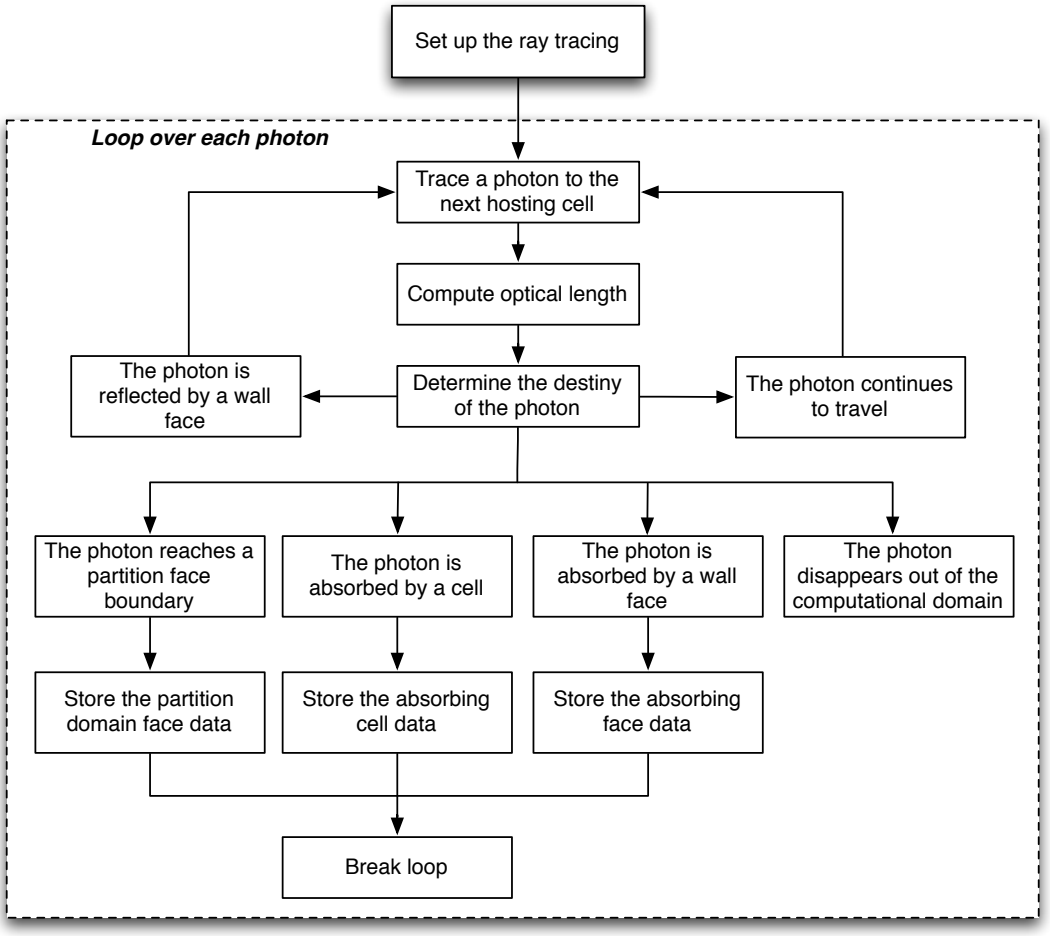
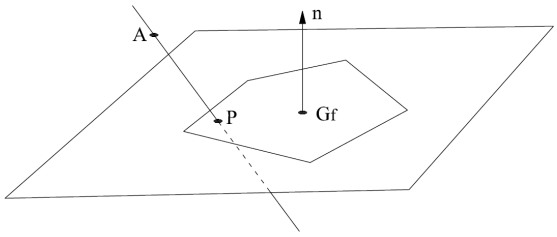


Figure 9.11: Flow Diagram of the Ray Tracing Program.

Another possibility different from the use of the double level approach, that is probably even more computationally efficient is to change the paradigm in which the jobs are divided between

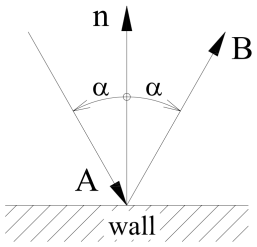
the processes. Instead of using the so called *domain decomposition*, one should use the *task decomposition*. In the first type the domain in subdivided into subdomains, and each process take care of one of this subdomain. This is the common paradigm used in CFD programs. In the second paradigm instead, each process deals with a specific task and have knowledge of what happens in the whole domain. In this case a task could be the particle tracking: so, the processes computes the trajectory of some particle along the whole domain therefore avoiding comunication. This is what is done in many computer graphic applications for the creations of movies. Unfortunatelly in CFD this approach is not the best suited, therefore the domain decomposition approach has been used. Now, two special considerations have to be devoted to how RayTracing takes into account the reflection, and how it makes the integration of optical length. Whenever a particle is reflected by a wall, the program calculates the coordinates of the impact point, see Fig. (9.12).



$$\begin{cases} n \cdot \overrightarrow{GfP} = 0 \\ P = A + m \cdot t \end{cases}$$

Figure 9.12: Computation of the reflection point P.

Then, from this point it reflects the reflected particle according to the specular rule, see Fig. (9.13). The integration of the optical length is performed with the rectangle rule: the most



$$\overline{B} = 2(\overline{A} \cdot \overline{n})\overline{n} - \overline{A}$$

Figure 9.13: Specular reflection rule.

suitable for this application; where each rectangle, i, is based on the particle trajectory segment that lies in cell i: see Fig. (9.14).

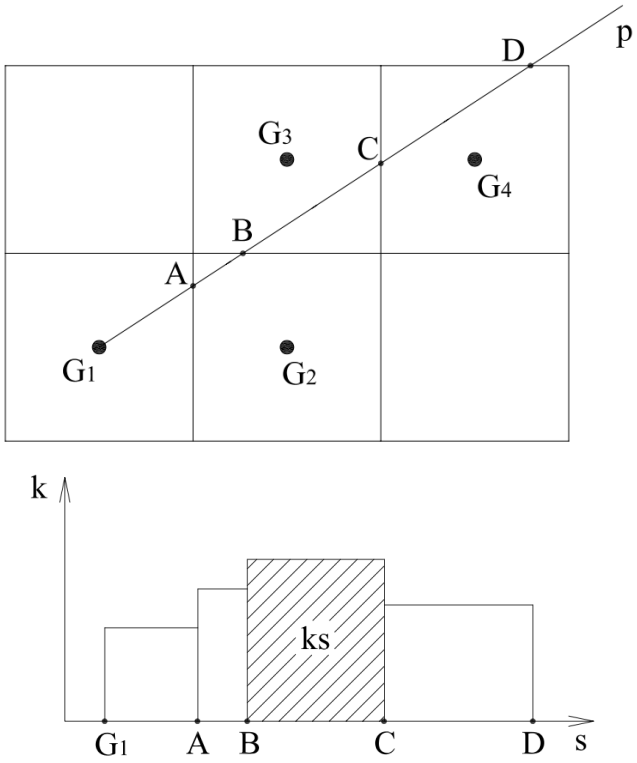


Figure 9.14: Integration of the Optical Length.

Theories have four stages of acceptance:
1) this is worthless nonsense;
2) this is an interesting, but perverse, point of view,
3) this is true, but quite unimportant;
4) I always said so.

J.B.S. Haldane

Chapter 10

Monte Carlo Method Implementation and Validation

We chose the solution method of READ, for the advantages that this method appears to have compared to the Energy method, see (8.4.7). Therefore the program is divided into two broad parts. In the first part all computations are done before the iteration loop on temperature and they are involved in calculating the coefficients $R(A, B)$ of the READ method. The second part takes care of calculating the radiative heat flux and is computed each time the temperature field changes, that is: for each time step. Here we will briefly discuss how these two parts have been implemented.
After we will show how the entire algorithm is validated against some analytical solution of 1D and 1D axialsimmetric problems [53, 91]. For the former we use the infinite slab model testcase, while for the latter we use the infinite cylinder testcase.

10.1 READ method implementation

The READ method is implemented in a READ function that acts as the master program for the parallelization of the particle tracking. The READ method has ultimately the purpose of associating to each element i (cell or face) a set of other elements j's (cells and / or faces) which individually absorb a number, $N_{i,j}$, of its energy particles. Therefore we have to implement a function f_{READ} defined as:

$$f_{READ} : \; i \rightarrow \; (j_i \times N_{i,j})^d \qquad i, j, N_{i,j}, d \in \mathbb{Z} \tag{10.1}$$

The number of entity d that absorb the energy particles emitted by element i is never greater than the number N_i of particle that i emits. In the READ method N_i is equal for all elements. This fact greatly facilitates the manipulation of vectors that store the collected datas. A priori, one can not know how many of these n particles are absorbed in the same processor that computes their emission and how many others are absorbed by the various other processors. We know however that with a high probability a good percentage of particles are absorbed

within the subdomain of the same processor that emits them. In general it can also happen that before a particle is absorbed it has crossed the subdomains of a number N_p of processors. Obviously a priori we do not know the N_p number of each particle of each element.
In order to have an implementation which, taking into account these comments, is as general as possible we have formulated the solution described in fig. (10.1). The implementation solution is composed of two modules: in the first one the processors compute the energy particle emitted inside the their subdomains; while in the second module the processes compute the energy particle that came in their subdomains from other partition subdomains.
The first module does a loop along all the energy particles emitted from all the elements inside the subdomains. It accomplishes this task calling the function *RayTracing* which can provide three different outputs depending on whether the particle is:

1. absorbed by a gas cell;
2. absorbed by a wall face;
3. end up in another subdomain;

moreover if the particle is dispersed it will not provide any output. In the first two cases, the acquired informations are stored in the vector R. In the third case instead the informations on the particle are stored in another vector P. Only for all energy particles already absorbed, the vector R stores information about the quantities i, j_i and $N_{i,j}$. Instead for all the energy particles that go to another partition subdomain, the vector P stores information on i and on the actual optical length. Once this loop is finished, the second module can start its procedure. First of all, the vector P is reorganized in a send buffer vector suitable to perform a total exchange with *MPI Alltoallv*. This reorganization is possible because based on the ID of the partition face we already know, thanks to *boundaryKnowledge* function, to which process a particle as to be communicated. Let's observe that if all processors have null sendbuffer size, then all particles have been computed till their absorption or dispersion; in this case the READ function has finished is job. If this is not the case then each processor has to compute, with the *RayTracing* function, the new energy particles that came from its partition subdomain border. Again the cases are: either that a energy particles is absorbed into this subdomain or that it go to another subdomain. In the first case the acquired informations are stored in a vector R_p which similar to R. Instead in the second case we rebuild the vector P with the new energy particles that go to another partition and we iterate the second module procedure. The loop of this procedure will stop when all processors have null sendbuffer size. In this case then the last task will be to do a total exchange of vector R_p and then merge the received R_p with R. Therefore at the end the vector R realizes the function (10.1).
One of the task of *BoundaryKnowledge* function is to build a map for each processor that has as key the partition face ID and as value the ID of the other processor that shares the partition face. This task is accomplished in a elegant way using the *MPI sendrecv* function to make a ring communication protocol between all the processors.

10.2 Implementing the computation of radiative heat flux

Once we use the read method, the part of the calculation of the radiative heat flux becomes simple. As shown in Fig. (10.2), the steps are:

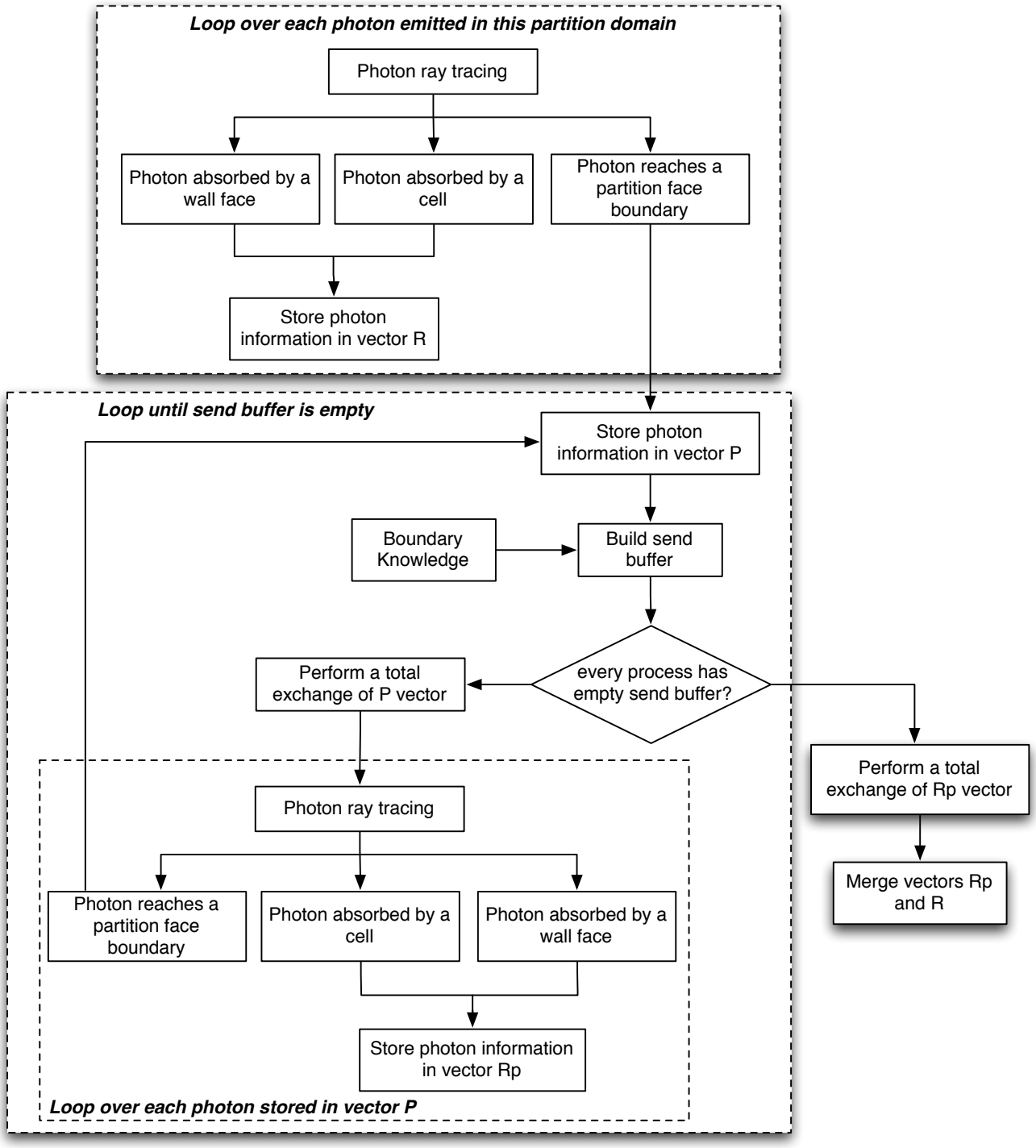


Figure 10.1: Flow diagram of the Monte Carlo implementation

1. compute the energy emitted by each element;
2. compute the energy carried by each element particle;
3. thanks to vector R build the sendbuffer with the fractions of energy absorbed by the various elements;
4. perform an MPI total exchange of send buffer: each processor sends a specific buffer of data to each other processor;
5. use the receive buffer to calculate the net radiative heat flux entering in each element;
6. and finally compute the net radiative heat flux in each element.

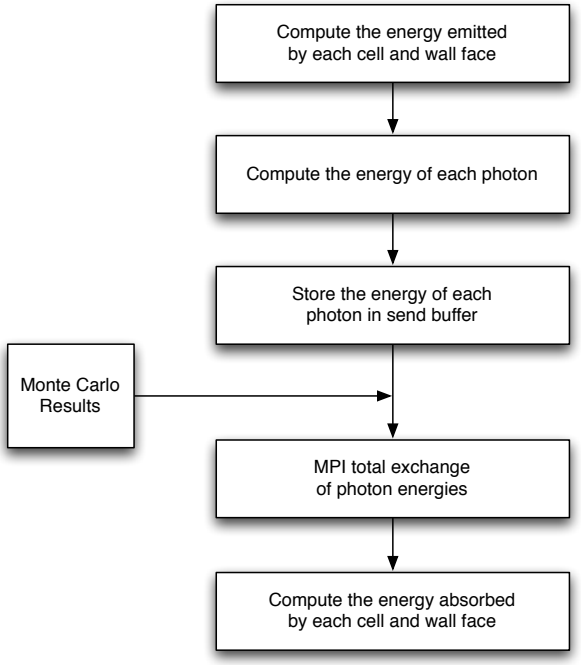


Figure 10.2: Flow diagram of the computation of radiative heat flux

10.3 Slab Testcase

10.3.1 Analytical problem

For an infinite slab there exists an analytical solution to compute the radiative heat flux. In ([61]) the analytical solution is derived. At the boundary $i+1$ of an infinite slab i the radiative heat flux q_{i+1} is:

$$q_{i+1} = q_i 2E_3(k) + \sigma T_m^4 \left(1 - 2E_3(k)\right) \tag{10.2}$$

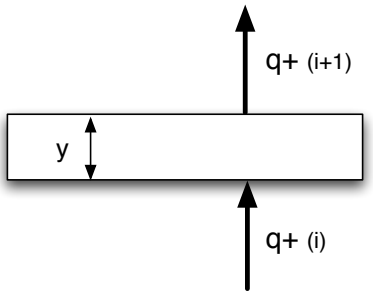


Figure 10.3: Single Slab model

where k is the optical thickness which for the slab it is simply definded as the product of the absorption coefficient, α, and the slab height:

$$k = \alpha k \tag{10.3}$$

σ is the Stefan-Boltzmann constant, equal to: $5.67051\dot{1}0^{-8} \quad Wm^{-2}K^{-4}$. E_3 is the exponential integral function defined as:

$$E_3(z) = \int_0^1 \mu e^{-\frac{z}{\mu}} \, d\mu \tag{10.4}$$

In this way we are able to analyze a slab with constant propriety along its thickness, mainly: constant temperature and absorption coefficient. But if we want to analyze a layer of fluid with variable property we should model it as a stack of slabs: each one with its own property values. Considering this model, now we have that on each slabs's boundary the heat flux is due to the contribution of the two slabs surrounding it. Therefore, for each boundary, $i+1$, we should compute two heat fluxes: one going up, q^+_{i+1}, and one going down, q^-_{i+1}, by equation (10.2), and then sum up these values:

$$\vec{q}_{i+1} = \vec{q^+}_{i+1} + \vec{q^-}_{i+1} \tag{10.5}$$

10.3.2 Numerical problem

A parallelepiped of dimensions: height ($H = 1m$), depth ($D = 1m$), length ($L = 10m$) has been simulated . The temperature distribution is linear along the height, from 0 K ($H = 0m$) to 10000 K ($H = 1m$). The absorption coefficient is constant throughout the volume equal to 1 m^{-1}. The discretization consists of 100 grid points along L, 100 along H, and a long D. In fact, the slabs are considered as stacked along H. The boundary surfaces that are coplanar with H, are considered walls that reflect the photons. The other two surfaces allow photons to escape out of the computational domain. Each cell emitted 300 particles.

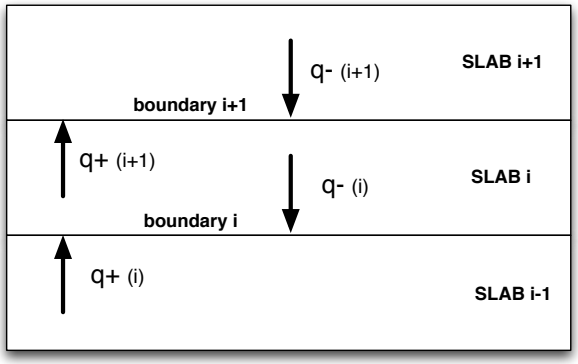


Figure 10.4: Stack of Slab model

10.3.3 Results

In fig. (10.5) are shown the analytical and Monte Carlo results for this problem. We can see a good agreement between the two results. We can also observe a bit of "noise" in the Monte Carlo one, probably due to the low number of particles per cell used and the low number of cells along the height: that is slabs.

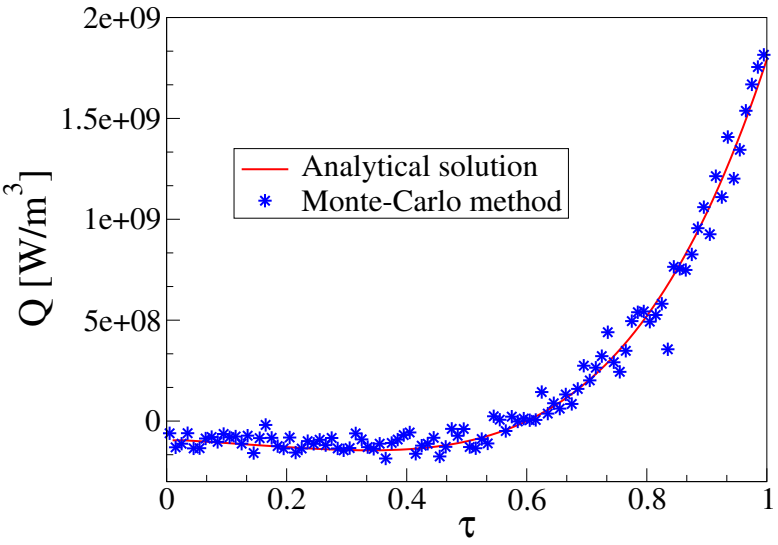


Figure 10.5: Slab testcase: Radiative source term (divergence of heat flux): analytical result (continuous line), Monte Carlo result (dotted line with points)

10.4 Cylinder Testcase

10.4.1 Analytical problem

For the infinite cylinder an analytical solution is given in ([100]), where a general expression for axisymmetric geometry is presented. For the case of a cylinder with non-emitting black

boundaries we have:

$$\overrightarrow{q^+}(r) = 4\int_0^{\pi/2}\left[\int_0^{r\cos\gamma}\alpha(r^I)e(r^I)D_2\left(\int_y^{r\cos\gamma}\alpha(r^{II})\,dy^I\right)\,dy\right]\cos\gamma d\gamma + 4\int_0^{\pi/2}\left[\int_0^{\sqrt{R^2-r^2\sin^2\gamma}}\alpha(r^I)e(r^I)D_2\left(\int_0^{r\cos\gamma}\alpha(r^{II})\,dy^I+\int_0^{y}\alpha(r^{II})\,dy^I\right)\,dy\right]\cos\gamma d\gamma \tag{10.6}$$

$$\overrightarrow{q^-}(r) = -4\int_0^{\pi/2}\left[\int_{r\cos\gamma}^{\sqrt{R^2-r^2\sin^2\gamma}}\alpha(r^I)e(r^I)D_2\left(\int_{r\cos\gamma}^{y}\alpha(r^{II})\,dy^I\right)\,dy\right]\cos\gamma d\gamma \tag{10.7}$$

where:

$$y = \sqrt{r^{I2} - r^2\sin^2\gamma} \tag{10.8}$$

$$y^I = \sqrt{r^{II2} - r^2\sin^2\gamma} \tag{10.9}$$

$$D_2(z) = \int_0^1 \frac{\mu}{\sqrt{1-\mu^2}} e^{-\frac{z}{\mu}}\,d\mu \tag{10.10}$$

The overall heat flux in radial direction, $q_{rad}(r)$, is then the sum of equation (10.6) and (10.7):

$$\vec{q}_{rad}(r) = \overrightarrow{q^+}(r) + \overrightarrow{q^-}(r) \tag{10.11}$$

$\vec{q}_{rad}(r)$ is the radial heat flux in J/m^2 at radial position r. Positive value means heat flux going outward from the centreline.

10.4.2 Numerical problem

The radiative heat flux along the radius of a cylinder of radius $r = 1m$ and length $l = 10m$ has been simulated . The temperature and absorption coefficients were constant along the radius and equal to 10000 K and 1 m^{-1} respectively. The reflective surfaces were these circular sections. The external cylindrical surface, allowed the photons to escape out of the computational domain.
The computing grid used had the same number, N, of grid points across the radius, length and a quarter of circumference. The discretization of the circular section was performed with a radial mesh. Two simulations were performed, one with N = 8 and 16 emitted particles per cell, and the second with N = 16 and 32 particles per cell.

10.4.3 Results

In Fig.s (10.6) and (10.7) are shown the results for the cylinder with a mesh of $N = 8$ and $N = 16$, respectively.

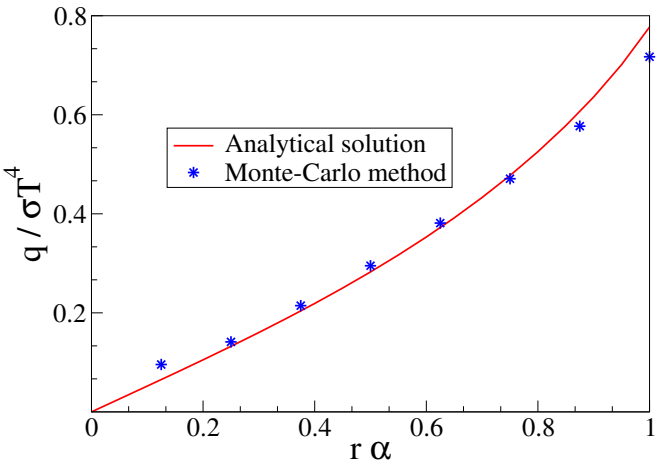


Figure 10.6: Cylinder testcase: Radiative heat flux from Monte Carlo Method and the Analytic solution with $N = 8$.

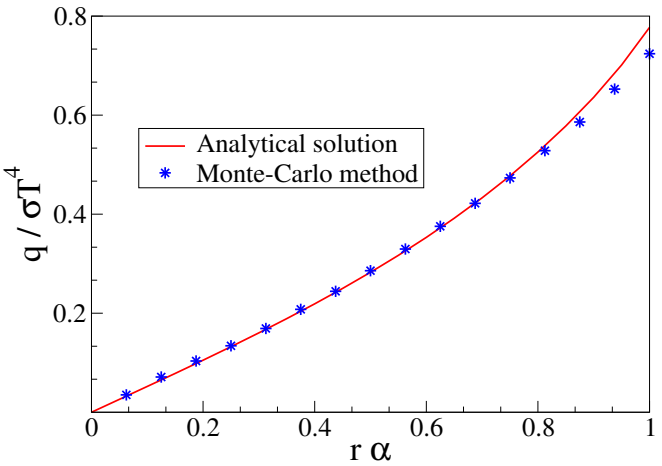


Figure 10.7: Cylinder testcase: Radiative heat flux from Monte Carlo Method and the Analytic solution with $N = 16$.

We can see a good agreement between the analytical and numerical solutions. Moreover we see that increasing the number N the error between the two decreases.

Part IV

Conclusion and Bibliography

Scientists study the world as it is,
Engineers create the world that has
never been.

Theodore von Karman

Chapter 11

Conclusion

In this last chapter, the achievements of the proposed solutions, the contributions to the field of study as well as some recommendations for future work are discussed.

11.1 Contributions of the Thesis

This thesis had a three-fold aim:

- The development and implementation of a multiphysics and multisolver computational environment.
- The development and implementation of a state of the art fluid flow solver.
- The development and implementation of a method for radiative heat transfer modeling and simulation.

We have devoted a part of this thesis to each of these aims, where they have been described. The accomplishment of this three-fold aim has required the definition and development of new strategies, which were not yet addressed in the literature. In this regard we shall now summarize the most significant achievements we have presented in our thesis; we do this following the list chapters.

Part I - Multiphysics Framework

Chapter 2- High Performance Computing In this chapter we have formulated our approach in order to accomplish high performance computing simulations. First of all we have explained the choice of programming language, parallel library and external library for linear system solving. Then we have described the programming methodology we consider to be the most appropriate for high performance computing. It consists on the extensive use of object-oriented programming, generic programming, and module or component based organization of the software tools. Moreover we have described two fundamental software tools which we have developed in order to make a dynamic use of computer memory and configure the

simulation parameters. These two tools have found large use in all software module we have implemented.

Chapter 3- Data Structure In this chapter the development and implementation of a parallel data structure for mesh based simulation has been addressed. The main concepts introduced here are those of:

1. *A Topology-Based Mesh Data Structure.* Which allows an effective description of the topology and the geometric model of the computational domain: all mesh entity related data are stored in a single object which is connected to the others through connectivity lists.

2. *The Flexible Mesh Representation.* Which allows to dynamically change the type of mesh representation according to the application needs. One of the most important features of this development is the ability to dynamically change the list of connectivity between the mesh entities.

3. *Mesh Entity Containers.* We have introduced the use of associative arrays for dynamically store the mesh entities. This type of data structure allow efficient insertion and deletion of elements, so that we are able to modify the mesh in any perspective.

4. *Field Variable Containers.* We have developed a special type containers for storing the dependent variable values. this type of data structure has all its elements stored in a continuous chunk of memory so that it is cache friendly. With it we can easilly interface our code with the external linear system solving libraries.

5. *The Space Regions.* The computational domain is divided in space regions, each one defined by a group of mesh entities. A space region allows to define regions with common physical property, or where a particular algorithm has to be applied. A space region can also define a boundary part of the domain where a particular boundary condition has to be applied.

6. *Management of Parallelization.* We have developed a robust and extremely general approach for managing the parallelization of the computation. It allows to transparently implement application's algorithm without care of parallel issues: all processes' managements are appropriately hidden. The management of the parallelization can be higly customizable, in order to meet the application requirements. For the migration of the data between processes we have developed a new approach which permits to transmit entire objects as they are, without issues related to their internal complexity.

7. *Interface with Application Code.* We have developed what we call *entity iterators*, which are special object whose aim is to act as a proxy for the mesh entity and transparently move from one entity to another, without care of the underling data structure.

8. *The Data Holder.* All the mesh data structures and its related algorithms are encapsulated inside a single object which can be managed dynamically. We called the class of this object as a *Data Holder*. In the same simulation we can have as many instances as we need of the Data Holder, each one devoted to a particular computational domain. Moreover the Data

Holder objects can share boundary information with each other, allowing the coupling between computational domain of different solvers.

9. *Geometric Variables.* At the end of the chapter we have reviwed some suitabe formulas for computing the volume of arbitrary shaped cells, and the area of arbitrary shaped faces.

Chapter 4- Solvers Framework In this chapter an innovative solvers' framework has been developed. Its main features are here summarized:

1. *Solvers' Framework Structure.* A general and efficient object oriented mechanism inside which computational modules of every kind can be dynamically plugged and connected to other modules.
2. *Splitting physics from numerics.* A mechanism for splitting the implementation and use of numerical methods from the physical model they want to solve.
3. *Callback Function.* A mechanism to dinamically perform indirect calls between the computational modules.
4. *Abstract Space Method Interface.* Each concrete space discretization method interfaces with the rest of the solver through the same interface which is managed dynamically in order that the concrete space method can be set or reset dynamically.
5. *Abstract Time Method Interface.* As for the space methods, also each concrete time discretization method interfaces with the rest of the solver through the same interface which is managed dynamically in order that the concrete time method can be set or reset dynamically.
6. *Abstract Linear Solver Interface.* The external libraries for solving linear system are interfaced with a dynamically allocable interface.
7. *Boundary Conditions.* We implemented a mechanism to associate a boundary condition to one or more space region. In each boundary condition the variables can be set with a polynomial of order up to the third. Moreover, as for almost everything in the Hydra programs, the end-user can higly customize the implementation of the boundary conditions in order to meet his/her needs.
8. *Automatic Differentiation.* We have introduced an automatic differentiation capability, through the development of a suitable tool. With these tools we are able to do things such as the automatic (and exact) computation of the Residual Jacobian. In order to use this tools all the classes or function of the space discretization module have been templatized. This task has required extreme care in order to be accomplished in an effective way.
9. *Physical Model.* Everything regarding the physics of the simulated phemomena are hidden in the physical model class, there is nothing related to a specific physical model inside the numerical modules, such as the time and space discretization ones. Apart from being elegant this approach has permitted to enable the splitting between physics from numerics.

Part II - Fluid Flow Modeling and Simulation

Chapter 5- Fluid Flow Modeling In this chapter we have reviewed the mathematical description of the fluid flow phenomena. This description is needed in order to implement a fluid flow solver. We have then described the issue of the boundary condition, and the definition of some of them. Finally we have presented an update list of the numerical method nowadays at our disposal for solving the fluid flow equations.

Chapter 6- Fluid Flow Solver Development In this chapter we have defined an approach for building a fluid flow solver, which can be inscribed inside the family of finite volume methods. The conceptual steps in order to compute the space residual vector have been defined, and for each step we chose a set of numerical methods which, in our opinion, are among the best one can find in the literature. Hence we showed how in our approach all these single numerical methods have to work together in order to build a space discretization method. The final result is a numerical method able to solve the Euler and Laminar Navier-Stokes equation on mixed 3D unstructured meshes.
We have also proposed a new modified Roe type scheme, we called it as *AMRoe*, for the computation of the convective fluxes, which can work from supersonic regimes to low-Mach number regimes.
Moreover we have briefly described some principles that we have followed for the implementation of the space discretization method, such as the dynamical configuration of the numerical scheme used for each conceptual step, and the use of the multiphysics framework described in the first part of the thesis.

Chapter 7- Fluid Flow Solver Validation In this chapter several test cases have been performed in order to validate both the multiplysics framework and the finite volume solver. All the simulations are runned in parallel and make use of the automatic differentiation for the computation of residual Jacobian. The convective fluxes are computed using our AMRoe scheme. These testcases are compared with analytical (where possible) or expected results, and all shown a good agreement. This has a twofold meaning:

1. The Fluid Flow solver works properly. It then means that our AMRoe scheme coupled with automatic differentiation works.
2. The whole multiphysics framework *machine* works.

Part III - Radiative Heat Transfer Modeling and Simulation

Chapter 8- Radiative Heat Transfer Modeling In this chapter we have dealt with the definition of a modellization of the phenomenon of radiative heat transfer suitable for computer simulation. We have introduced a novel approach based on what is called READ method which can be inscribed inside the framework of the Monte Carlo methods. The advantage of our approach consists in performing all the computations needed to determine the statistical pattern of the computational domain before the iterative time marching procedure. Inside each iteration of the time marching procedure remains only the function for the computation of the radiative heat transfer based on the energy emmited from each mesh cell and wall boundary

face. Our approach has demonstrated to be competitive with the one developed by the DLR (the german space agency).

Chapter 9- Particle Tracking In order to determine the statistical pattern of the computational domain we must trace each photon emitted by the mesh cell and wall boundary face accross all the computational domain untill it is absorbed by onother entity. In order to do this we have developed a novel particle tracking algorithm based on the work of Chorda, which is able to trace a particle over any mixed unstructured 2D or 3D meshes. We have managed to implement this algorithm in parallel in a general fashion so that it can be used not only for thace photon but also any kind of entity that moves inside the computational domain, such as: solid particle, bubble and virtual entity.

Chapter 10- Monte Carlo Method Implementation and Validation In this chapter we have briefly described the algorithm we have developed for parallel implementing our Monte Carlo Method for Radiative Heat Transfer. After that we have shown two testcases that we used to validate the method. These testcases show the reliability and efficiency of the developed method.

11.2 Future Work

We shall now discuss some recommendations for future work, that we think could somehow *complete* the job.

Turbulence modeling Implement some turbulence models (see [92], [120] and [90]) like those of Spalart-Almaras, $k-\varepsilon$, $k-\omega$, and Baldwin Lomax. With these turbulence models we would be able to deal with the Turbulence Navier-Stokes equations and simulate most of the problems encountered in industrial applications.

Arbitrary Lagrangian Eulerian formulation Create a solver for Arbitrary Lagrangian Eulerian Formulation of finite volume method in order to be able to deal with moving boundary problem, that is, those problems which arise when the solid wall undergo roto-translational movement and/or shape-modification. There are very few codes in the market which are adequately able to takle this type of simulation, due to the difficulty to solve challenges that the code implementation arises.

Finite Element Solver for solid mechanics Very interesting type of simulations are those of fluid structure interaction. This simulations require four ingredients:

- A flow solver with arbitrary lagrangian eulerian formulation of the numerical scheme.
- A mesh moving tools.
- A finite element solver for the solid domain.

- A coupling strategy between the fluid and the solid computation.

Fluid structure interaction problems are gaining more and more interest in both academia and industry, therefore would be advantageous to be able to offer a software tools for their treatment.

Mesh generator and modification tool We have just mentioned that for fluid-structure interaction problem we need a mesh modification tool. More in general we want to suggest the development of a software module for unstructured 3D mesh construction and modification.

Bibliography

[1] BLAS. http://www.netlib.org/blas/.

[2] Boost c++ library. http://www.boost.org/.

[3] CFD open software list and much more. http://www.cfd-online.com/Links/soft.html#cfd.

[4] Fenics. http://fenicsproject.org/.

[5] Gambit. http://www.ansys.com/Products/Simulation+Technology/Fluid+Dynamics/ANSYS+FLUENT.

[6] Hydra page from wikipedia. http://en.wikipedia.org/wiki/Lernaean_Hydra.

[7] Lifev. http://www.lifev.org/.

[8] Openflower. http://openflower.sourceforge.net/index2.html.

[9] Overture. https://computation.llnl.gov/casc/Overture/.

[10] Paraview. http://www.paraview.org/.

[11] Sacado. http://trilinos.sandia.gov/packages/sacado/faq.html.

[12] Tecplot. http://www.tecplot.com/.

[13] UML. http://www.uml.org/.

[14] VTK. http://www.vtk.org/.

[15] Zoltan. http://www.cs.sandia.gov/Zoltan/.

[16] A. Alexandrescu. *Modern C++ Design*. Addison-Wesley, 2001.

[17] J.D. Anderson. *Hypersonic and High Temperature Gas Dynamics*. McGraw-Hill, 1989.

[18] J.D. Anderson. *Modern Compressible Flow*. McGraw-Hill, 1990.

[19] J.D. Anderson. *Introduction to Computational Fluid Dynamics*. von Karman Institute Lecture Series, 2009.

[20] J.D. Anderson. *Fundamentals of Aerodynamics*. McGraw-Hill, 2010.

[21] W.K. Anderson and D.L. Bonhaus. An implicit upwind algorithm for computing turbulent flows on unstructured grids. *Computers Fluids*, 23(1):1–21, 1994.

[22] T.J. Barth. A 3-d upwind euler solver for unstructured meshes. *AIAA paper*, (91-1548), 1991.

[23] T.J. Barth. Recent developments in high order k-exact reconstruction on unstructured meshes. *AIAA paper*, (93-0668), 1993.

[24] T.J. Barth and P.O. Frederickson. Higher order solution of the euler equations on unstructured grids using quadratic recontruction. *AIAA paper*, (90-0013), 1990.

[25] T.J. Barth and D.C. Jespersen. The design and application of upwind schemes on unstructured meshes. *AIAA paper*, (89-0366), 1989.

[26] W.H. Beall and M.S. Shephard. A general topology-based mesh data structure. *International Journal of Numerical Methods in Engineering*, 40(1573), 1997.

[27] R.B. Bird, W.E. Stewart, and E.N. Lightfoot. *Transport phenomena*. John Wiley and Sons, Inc., 2007.

[28] Z. Chen. *Finite Element Methods and Their Applications*. Springer, 2005.

[29] R. Chorda, J.A. Blasco, and N. Fueyo. An efficient particle-locating algorithm for application in arbitrary 2d and 3d grids. *International Journal of Multiphase Flow*, 28:1565–1580, 2001.

[30] H.M. Deitel and P.J. Deitel. *C++ How to Program*. Prentice-Hall, Inc., 2005.

[31] M. Delanaye. *Polynomial Reconstruction Finite Volume Schemes for the Compressible Euler and Navier-Stokes Equations on Unstructured Adaptive Grids*. PhD thesis, University of Liege, 1996.

[32] M. Delanaye and J.A. Essers. Finite volume scheme with quadratic recontruction on unstructured adaptive meshes applied to turbomachinery flows. *ASME IGTI Gas Turbine Conference*, 1995.

[33] S. Dellacherie. Analysis of godunov type schemes applied to the compressible euler system at low mach number. *Journal of Computational Physics*, 229:978–1016, 2010.

[34] J. Donea and A. Huerta. *Finite Element Methods for Flow Probelms*. John Wiley and Sons, 2003.

[35] P. Dutt. Stable boundary conditions and difference schemes for navier-stokes equations. *SIAM Journal on Numerical Analysis*, 25(2), 1988.

[36] J.R. Edwards and M.S. Liou. Low-diffusion flux-splitting methods for flows at all speeds. *AIAA Journal*, 36:1610–1617, 1998.

[37] H.W. et al. Openfoam: The open source cfd toolbox. http://www.opencfd.co.uk/openfoam, 2004.

[38] John W. Eaton et. al. Octave. http://www.gnu.org/software/octave/.

[39] J.H. Ferziger and M. Peric. *Computational Methods for Fluid Dynamics*. Springer, 2002.

[40] J.E. Flaherty. *Finite Element Analysis*. 2000.

[41] Fletcher. *Computational Thecnique for Fluid Dynamics*. Springer-Verlag, 1991.

[42] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[43] P. Garabedian. *Partial Differential Equations*. Wiley, 1964.

[44] R.V. Garimella. Mesh Data Structure Selection for Mesh Generation and FEA Applications. *International Journal of Numerical Methods in Engineering*, 55(4), 2002.

[45] C. Grossmann and H.G. Roos. *Numerical Treatment of Partial Differential Equations*. CRC Press, 2007.

[46] H. Guillard and C. Viozat. On the behaviour of upwind schemes in the low mach number limit. *Computers and Fluids*, 28:63–86, 1999.

[47] B. Gustafsson and A. Sundstrom. Incompletely parabolic problems in fluid dynamics. *SIAM Journal on Applied Mathematics*, 35(2), 1978.

[48] A. Haselbacher and J. Blazek. On the accurate and efficient discretization of the navier-stokes equations on mixed grids. *AIAA journal*, 1999.

[49] C. Hirsh. *Numerical Computation of Internal and External Flows, Computational Methods for Inviscid and Viscous Flows*. John Wiley and Sons, Inc., 1990.

[50] C. Hirsh. *Numerical Computation of Internal and External Flows, Fundamentals of Computational Fluid Dynamics*. Elsevier, 2007.

[51] T.J.R. Hughes. *The Finite Element Method*. Prentice-Hall, Inc., 1987.

[52] A. Jameson. Analysis and design of numerical schemes for gas dynamics, 2: Artificial diffusion and discrete shock structure. *International Journal of Computational Fluid Dynamics*, 5:1–38, 1995.

[53] S. Karl. Simulation of radiative effects in plasma flows. VKI 18, Von Karman Institute, June 2001.

[54] G. Karypis. Parmetis: Parallel mesh partitioning. http://www-users.cs.umn.edu/~karypis/metis/parmetis.

[55] B. Kirk, J.W. Peterson, R.H. Stogner, and G.F. Carey. Libmesh: A c++ library for parallel adaptive mesh refinement/coarsening simulations. http://dx.doi.org/10.1007/s00366-006-0049-3.

[56] P.K. Kundu and I.M. Cohen. *Fluid Mechanics*. Academic Press, 2002.

[57] Argonne National Laboratories. Petsc: Portable, extensible toolkit for scientific computation. http://www-unix.mcs.anl.gov/petsc.

[58] Sandia National Laboratories. The trilinos project. http://software.sandia.gov/trilinos.

[59] A. Lani. *An Object Oriented and High Performance Platform for Aerothermodynamics Simulation*. PhD thesis, von Karman Institute for Fluid Dynamics, 2008.

[60] C.L. Lawson, R.J. Hanson, D. Kincaid, and F.T. Krogh. Basic linear algebra subprograms for FORTRAN usage. *ACM Trans. Math. Soft.*, (5):308–323, 1979.

[61] H. Lee and R.O. Buckius. Reducing scattering to nonscattering problems in radiation heat transfer. *International Journal of Heat and Mass Transfer*, 26(7):1055–1062, 1983.

[62] R.L. LeVeque. *Finite Volume Methods for Hyperbolic Problems*. Cambridge University Press, 2004.

[63] R.L. LeVeque. *Finite Difference Methods for Ordinary and Partial Differential Equations*. SIAM, 2007.

[64] B.Q. Li. *Discontinuous Finite Elements in Fluid Dynamics and Heat Transfer*. Springer, 2006.

[65] G. Li and M.F. Modest. An effective particle tracing scheme on structure/unstructured grids in hybrid finite volume/pdf monte carlo methods. *Journal of Computational Physics*, 173:187–207, 2001.

[66] X.S. Li and C.W. Gu. An all-speed roe-type scheme and its asymptotic analysis of low-mach number behaviour. *Journal of Computational Physics*, 227:5144–5159, 2008.

[67] X.S. Li, C.W. Gu, and J.Z. Xu. Development of roe-type scheme for all-speed flows based on preconditioning method. *Computers and Fluids*, 38:810–817, 2009.

[68] H.M. Liebernstein. *Theory of Partial Differential Equations*.

[69] M.S. Liou. On a new class of flux splittings. In *Proc. 13th Int. Conf. on Numerical Methods in Fluid Dynamics*, Rome, Italy, 1992.

[70] M.S. Liou. Progress towards an improved cfd method - ausm+. *AIAA paper*, 95-1701, 1995.

[71] M.S. Liou. A sequel to ausm: Ausm+. *Journal of Computational Physics*, 129:364–382, 1996.

[72] M.S. Liou. A sequel to ausm, part ii: Ausm+up for all speeds. *Journal of Computational Physics*, 214:137–170, 2006.

[73] M.S. Liou and C.J. Steffen. A new flux splitting scheme. *Journal of Computational Physics*, 107:23–39, 1993.

[74] G.R. Liu. *Mesh Free Methods*. CRC Press, 2003.

[75] R. Lohner and J. Ambrosiano. A vectorized particle tracer for unstructured grids. *Journal of Computational Physics*, 91:22–31, 1990.

[76] H. Lomax, T. Pulliam, and D. Zingg. *Fundamental of Computational Fluid Dynamics*. Springer, 2003.

[77] MathWorks. Matlab - the language of technical computing. http://www.mathworks.nl/products/matlab/index.html.

[78] M. Matsumoto and T. Nishimura. Mersenne twister: A 623-dimensionally equidistributed uniform pseudo-random number generator. *ACM Transactions on Modeling and Computer Simulation*, 8(1):3–10, 1998.

[79] D.J. Mavriplis. Three dimensional multigrid reynolds averaged navier stokes solver for unstructured meshes. *AIAA paper*, (94-1878), 1994.

[80] S. Meyers. *More Effective C++*. Addison-Wesley, 1996.

[81] S. Meyers. *Effective C++*. Addison-Wesley, 1998.

[82] C.R. Mitchell. Improved recontruction schemes for the navier-stokes equations on unstructured meshes. *AIAA paper*, (94-0642), 1994.

[83] C.R. Mitchell and R.W. Walters. K-exact reconstruction for the navier-stokes equations on arbitrary grids. *AIAA paper*, (93-0536), 1993.

[84] M.F. Modest. Backward monte carlo simulations in radiative heat transfer. *Journal of Heat Transfer*, 125, 2003.

[85] M.F. Modest. *Radiative Heat Transfer*. 2003.

[86] J. Oliger and A. Sundstrom. Theoretical and practical aspects of some initial-boundary value problems in fluid dynamics. *STANFORD UNIVERSITY*, STAN-CS-76-578, 1976.

[87] E.F. Owzarek. *Fundamentals of Gas Dynamics*. International Textbook Company, 1968.

[88] P.S. Pacheco. *Parallel Programming with MPI*. Morgan Kaufmann Publishers, 1997.

[89] R. Peyret. *Spectral methods for incompressible viscous flow*. Springer, 2002.

[90] S.B. Pope. *Turbulent Flows*. Cambridge University Press, 2000.

[91] D. Potter and S. Karl. Validation of aerothermo-chemistry models for re-entry applications. AMOD TN 3.2, ESA, June 2009.

[92] P.R.Spalart and S.R. Allmaras. A one-equatlon turbulence model for aerodynamic flows. *AIAA paper*, (92-0439), 1992.

[93] T. Quintino. *A Component Environment for High-Perormance Scientific Computing*. PhD thesis, von Karman Institute for Fluid Dynamics, 2008.

[94] M. Ricchiuto. *Construction and Analysis of Compact Residual Discretizations for Conservation Laws on Unstructured Meshes*. PhD thesis, von Karman Institute for Fluid Dynamics, 2005.

[95] B. Riviere. *Discontinuous Galerkin Methods for Solving Elliptic and Parabolic Equations*. Siam, 2008.

[96] P.L. Roe. Approximate riemann solvers, parameter vectors, and difference schemes. *Journal of Computational Physics*, 43:357–372, 1981.

[97] W.K. Liu S. Li. *Meshfree particle methods*. Springer, 2004.

[98] Yousef Saad. *Iterative methods for sparse linear systems*. Siam, 2003.

[99] Yousef Saad. *Numerical Methods for Large Eigenvalue Problems - 2nd Edition*. Siam, 2011.

[100] K. Sawada, T. Sakai, and M. Mitsuda. Application of planck-rosseland-gray model for high entalpy arc heaters. *AIAA Paper*, 98-0861, 1998.

[101] J.J. Smolderen. *Partial Differential Equations and Their Numerical Solutions*. von Karman Institute course note.

[102] M. Snir, S. Otto, S.H. Ledermann, D. Walker, and J. Dongarra. *MPI: The Complete Reference*. MIT Press, 1996.

[103] P. Solin. *Partial Differential Equations and the Finite Element Method*. John Wiley and Sons, Inc., 2006.

[104] B. Stroustrup. *The C++ Programming Language*. Addison Wesley, 2000.

[105] J.F. Thompson, B.K. Soni, and N.P. Weatherill. *Handbook of Grid Generation*. CRC Press, 1999.

[106] B. Thornber and D. Drikakis. Numerical dissipation of upwind schemes in low mach flow. *International Journal for Numerical Methods in Fluids*, 56:1535–1541, 2008.

[107] E.F. Toro. *Riemann Solvers and Numerical Methods for Fluid Dynamics*. Springer, 1999.

[108] A. Toselli and O. Widlund. *Domain Decomposition Methods – Algorithms and Theory*. Springer, 2005.

[109] E. Turkel. Preconditioning techniques in computational fluid dynamics. *Annual Reviews of Fluid Mechanics*, 31:385–416, 1999.

[110] A. Tveito and R. Winther. *Introduction to Partial Differential Equations: A Computational Approach*. Springer, 1998.

[111] Guido van Rossum et. al. Python. http://www.python.org/.

[112] D. Vandervoorde and N.M. Josuttis. *C++ Templates, The Complete Guide*. Addison-Wesley, 2010.

[113] V. Venkatakrishnan. On the accuracy of limiters and convergence to steady state solutions. *AIAA paper*, (93-0880), 1993.

[114] V. Venkatakrishnan. Convergence to steady state solutions of the euler equations on unstructured grids with limiters. *Journal of Computational Physics*, 118:120–130, 1995.

[115] V. Venkatakrishnan and D.J. Mavriplis. Implicit solvers for unstructured meshes. *Journal of Computational Physics*, 105:83–91, 1993.

[116] V. Venkatakrishnan and D.J. Mavriplis. Implicit method for the computation of unsteady flows on unstructured grids. *Journal of Computational Physics*, 127:380–397, 1996.

[117] Y. Wada and M.S. Liou. A flux splitting scheme with high-resolution and robustness for discontinuities. *AIAA paper*, 94-0083, 1994.

[118] J.M. Weiss and W.A. Smith. Preconditioning applied to variable and constant density flows. *AIAA Journal*, 33:2050–2057, 1995.

[119] M. Widhalm, C. Bartels, J.Meyer, and N. Kroll. Lagrangian particle tracking on large unstructured three-dimensional meshes. *AIAA Aerospace Science Meeting and Exhibit*, 472, 2008.

[120] D.C. Wilcox. *Turbulence Modeling for CFD*. DCW Industries, Inc., 1994.

[121] Y. Wu, M.F. Modest, and D.C. Haworth. A high-order photon monte carlo method for radiative transfer in direct numerical simulation. *Journal of Computational Physics*, 224:898–922, 2007.

[122] W.J. Yang, H. Taniguchi, and K. Kudo. *Radiative Heat Transfer by the Monte Carlo Method*. Academic Press, 1995.

[123] Q. Zhou and M.A. Leschziner. An improved particle-locating algorithm for eulerian-lagrangian computations of two-phase flows in general coordinates. *International Journal of Multiphase Flow*, 25:813–825, 1999.

[124] O.C. Zienkiewicz and R.L. Taylor. *The Finite Element Method, Volume3: Fluid Dynamics*. Butterworth Heinemann, 2000.

[125] O.C. Zienkiewicz and R.L. Taylor. *The Finite Element Method, Volume3: Solid Mechanics*. Butterworth Heinemann, 2000.

[126] O.C. Zienkiewicz and R.L. Taylor. *The Finite Element Method, Volume3: The Basis*. Butterworth Heinemann, 2000.